The Dissertation Committee for Dimitrios Kaseridis
certifies that this is the approved version of the following dissertation:

Memory-subsystem Resource Management for the Many-core Era

Committee:

Lizy Kurian John, Supervisor

Nur A. Touba

Derek Chiou

Jim Holt

Paul V. Gratz

Memory-subsystem Resource Management for the Many-core Era

by

Dimitrios Kaseridis, B.E

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2011

To my parents Vasilis and Despoina.

Acknowledgments

My thesis is dedicated to my loving parents, Vasilis Kaseridis and Despoina Kaseridou. Their guidance, encouragement, love and unconditional support are the main reasons in my completing of this PhD. They have always supported my decisions even if it required them to change the course of their life. I will always be grateful to them. In addition, I would like to thank my family and friends that I had to leave back in Greece to pursue this PhD. They always believed in me and supported my decisions.

I am grateful to my advisor, Lizy Kurian John, as she gave me the opportunity to work in a research topic that I enjoyed and allowed me to be as creative as I could. Lizy always provided the right balance of freedom and guidance that eventually helped my development as a researcher.

I thank Nur Touba, Derek Chiou, Jim Holt and Paul V. Gratz for serving in my PhD committee. Their constructive criticism helped improve the quality of this dissertation.

Special thanks to Jeff Stuecheli for being an excellent colleague, coauthor and a friend. Jeff was usually the first person to discuss research ideas and helped me transform many raw ideas into concrete research results.

Throughout the six years of my PhD studies, I had the opportunity to collaborate with a number of great people. First, I need to thank all of the members of the LCA research group for providing a creative and helpful environment for my studies. When I initially started at UT, I had the pleasure to collaborate and, later on, be a good friend with Elias Mizan, Aashish Phansalkar, Hari Angepat and Ciji Isen. They always showed continuous interest in my progress and were a constant source of encouragement. I also want to thank Jian Chen, Faisal Iqbal, Karthik Ganesan, Arun Nair, Umar Farooq and

Jungho Jo for their feedback during various stages of my work and for participating in some of our endless discussions about research, academia and life in general.

Finally, I would like to thank the organizations that supported me financially during my PhD studies. National Science Foundation (NSF), Semiconductor Research Corporation (SRC) and the University of Texas at Austin provided me grants/fellowships for my research. In addition, Intel Corporation and National Science Foundation donated the equipment on which I conducted most of my simulations.

Memory-subsystem Resource Management for the Many-core Era

Publication No. _____

Dimitrios Kaseridis, Ph.D.
The University of Texas at Austin, 2011

Supervisor: Lizy Kurian John

As semiconductor technology continues to scale lower in the nanometer era, the communication between processor and main memory has been particularly challenged. The well-studied frequency, memory and power "walls" have redirect architects towards utilizing Chip Multiprocessors (CMP) as an attractive architecture for leveraging technology scaling. In order to achieve high efficiency and throughput, CMPs rely heavily on sharing resources among multiple cores, especially in the case of the memory hierarchy. Unfortunately, such sharing introduces resource contention and interference between the multiple executing threads.

The ever-increasing access latency difference between processor and memory, the gradually increasing memory bandwidth demands to main memory, and the decreasing cache capacity size available to each core due to multiple core integration, has made the need for an efficient memory subsystem resource management more critical than ever before. This dissertation focuses on managing the sharing of the *Last-level Cache* (LLC) capacity and the *main memory bandwidth*, as the two most important resources that significantly affect system performance and energy consumption. The presented

schemes include efficient solutions to all of the three basic requirements for implementing a resource management schemes, that is: a) profiling mechanisms to capture applications' resource requirements, b) microarchitecture mechanisms to enforce a resource allocation scheme, and c) resource allocations algorithms/policies to manage the available memory resources throughput the whole memory hierarchy of a CMP system.

To achieve these targets the dissertation first describes a set of low overhead, non-invasive profiling mechanisms that are able to project applications' memory resource requirements and memory sharing behavior. Two memory resource partitioning schemes are presented. The first one, the *Bank-aware* dynamic partitioning scheme provides a low overhead solution for partitioning cache resources of large CMP architectures that are based on a *Dynamic Non-Uniform Cache Architecture* (DNUCA) last-level cache design, consistent with the current industry trends. In addition, the second scheme, the *Bandwidth-aware* dynamic scheme presents a system-wide optimization of memory-subsystem resource allocation and job scheduling for large, multi-chip CMP systems. The scheme is seeking for optimizations both within and outside single CMP chips, aiming at overall system throughput and efficiency improvements.

As cache partitioning schemes with isolated partitions impose a set of restrictions in the use of the last-level cache, which can severely affect the performance of large CMP designs, this dissertation presents a *Quasi-partitioning* scheme that breaks such restrictions while providing most of the benefits of cache partitioning schemes. The presented solution is able to efficiently scale to a significant larger number of cores than what previously described schemes that are based on isolated partition can achieve.

Finally, as the memory controller is one of the fundamental components of the memory-subsystem, a well-designed memory-subsystem resource management needs to carefully utilize the memory controller resources and coordinate its functionality with the operation of the main memory and the last-level cache. To improve execution fairness

and system throughput, this dissertation presents a criticality-based, memory controller requests priority scheme. The scheme ranks demand read and prefetch operations based on their latency sensitivity, while it coordinates its operation with the DRAM page-mode policy and the memory data prefetcher.

Table of Contents

Acknowledgments v

Abstract vii

List of Tables xv

List of Figures xvi

Chapter 1. Introduction 1

1.1 Summary of Challenges and Solutions 2
1.2 Thesis Statement 7
1.3 Contributions 7
1.4 Dissertation Organization 9

Chapter 2. Background Terminology and Related Work 10

2.1 Memory Subsystem Background 10
2.1.1 Memory Caches 10
2.1.2 Main Memory 13
2.1.3 Memory Controller 14
2.2 Related Work in Cache Management 15
2.2.1 Isolated Cache Partitions 15
2.2.2 Dead-time Management 16
2.2.3 Cache Pseudo-Partitioning 16
2.2.4 QoS/Fairness 17
2.2.5 Cache Replacement Policies 17
2.3 Related Work in Memory Controller Policies 18

Chapter 3. Simulation Methodology, Tools and Workloads **20**
3.1 Full System Timing Simulator 21
3.2 Detailed Microarchitecture Simulator 22
3.3 Benchmark Suites 24
3.3.1 SPEC CPU2006 24
3.3.2 SPEC JBB2005 25
3.3.3 SPLASH-2 25
3.3.4 Apache Web-server 26

Chapter 4. Profilers for Application's Memory-subsystem Resource Requirements **28**
4.1 Last-level Cache Capacity 29
4.2 Memory Bandwidth 30
4.3 Examples of Real Applications' Bandwidth Requirements 33
4.4 MSA-profiler Implementation Overhead and Accuracy 35

Chapter 5. Bank-aware Cache Partitioning for Multicore Architectures **40**
5.1 Introduction 41
5.2 CMP-Baseline 43
5.3 Bank-aware Cache Partitioning 46
5.3.1 Cache Profiling of Applications 46
5.3.2 Bank-aware Assignment of Cache Capacity 48
5.3.3 Allocation Algorithm on CMP 52
5.4 Evaluation 53
5.4.1 *Bank-aware* vs. *Unrestricted* Partitioning 54
5.4.2 Detailed Simulation Results 56
5.5 Summary 58

Chapter 6. Bandwidth-aware Memory-subsystem Resource Management for Large CMP Systems **60**
6.1 Introduction 61
6.2 Baseline System Architecture 63
6.3 Bandwidth-aware Resource Management 64
6.3.1 Applications' Resource Profiling 65

6.3.2 Intra-Chip Partitioning Algorithm 66
6.3.3 Inter-Chip Partitioning Algorithm 68
6.3.3.1 Cache Capacity 69
6.3.3.2 Memory Bandwidth 70
6.3.3.3 Computational Overhead of Inter-chip algorithm 72
6.3.4 Overall Dynamic Scheme 72
6.4 Evaluation 73
6.4.1 Last-level cache misses 75
6.4.2 Memory Bandwidth 78
6.4.3 Detailed Simulation – Multiprogrammed Case Study 79
6.4.4 Detailed Simulation – Multithreaded Case Study 81
6.5 Related Work 85
6.6 Summary 86

Chapter 7. High Performance Quasi-partitioned Last-level Caches 87
7.1 Introduction 88
7.2 Motivation 90
7.2.1 Memory-level Parallelism (MLP) 90
7.2.2 Cache Friendliness Aware Cache Allocation 91
7.2.2.1 Need for Cache Pseudo-partitioning Schemes 92
7.2.2.2 Applications Cache Behavior 92
7.3 MCFQ: MLP and Cache-friendliness aware Quasi-partitioning Scheme . . 95
7.3.1 Profiling Applications' Cache Demands, Average Concurrency Factor and Memory Behavior 96
7.3.2 Allocation of Cache Capacity based on Performance Sensitivity of Misses 98
7.3.2.1 Cache Capacity Allocation Algorithm 99
7.3.3 Cache-friendliness aware Quasi-Partitioning Scheme 101
7.3.3.1 Cache-line Insertion, Promotion and Replacement Policy . 101
7.3.3.2 Interference Sensitivity Factor and Partition Scaling Scheme 103
7.3.4 Overall dynamic Last-level cache Quasi-partitioning scheme 106
7.4 Evaluation 108
7.4.1 MLP-aware Capacity Assignment 110

7.4.2 MCFQ – Performance evaluation on a 4 & 8 cores CMP 110
7.4.3 MCFQ – Fairness evaluation on a 4 & 8 cores CMP 115
7.5 Summary 117

Chapter 8. A Criticality-based DRAM Memory Controller Scheme Based on a hybrid open/close Page-mode Policy 119
8.1 Introduction 120
8.2 Motivation 123
8.2.1 Balanced Designs and Importance of Cache Capacity and Memory Bandwidth 124
8.2.2 Benefits of DRAM Page-mode operation 125
8.2.3 Bank and Row Buffer Locality Interplay With Address Mapping . . 128
8.3 A Criticality-based DRAM Memory Controller Scheme Based on a hybrid open/close Page-mode Policy 131
8.3.1 DRAM Address Mapping Scheme 132
8.3.2 Data Prefetch Engine 133
8.3.2.1 Multi-line Prefetch Requests 134
8.3.2.2 Multi-line prefetches combined with address mapping scheme 134
8.3.3 Memory Request Queue Scheduling Scheme 135
8.3.3.1 DRAM Read Requests Priority Calculation 135
8.3.3.2 DRAM Page Closure (Precharge) Policy 137
8.3.3.3 Overall Memory Requests Scheduling Scheme 137
8.3.3.4 Handling Write Operations 138
8.4 Evaluation 139
8.4.1 Target Page-hit Count Sensitivity 142
8.4.2 Throughput 143
8.4.3 Page-hits per Page activation 145
8.4.4 Fairness 146
8.4.5 DRAM Energy Consumption 147
8.5 Summary 148

Chapter 9. Summary and Future Work **150**
9.1 Summary 150
9.2 Future Work 152
9.2.1 Multi-chip / multi-socket QoS scheme with capacity planning . . . 152
9.2.2 Extension of Resource Managing schemes for fairness/QoS 153
9.2.3 Bandwidth-aware page allocation memory controller policy 153

Bibliography **155**

List of Tables

4.1 Overhead of the presented MSA-based profiler. 38

5.1 DNUCA-CMP parameters used for evaluation of the Bank-aware scheme. 45
5.2 Overhead of the proposed MSA profiler. 47
5.3 8-core full system workload experiments. 57

6.1 Single-Chip CMP parameters used to evaluate the "Bandwidth-aware" scheme. 65
6.2 Heterogeneous Multithreaded Workloads. 83

7.1 Full-system detailed simulation parameters assumed for the evaluation of MCFQ. 107
7.2 Multi-programed benchmark sets from SPEC CPU2006 [16] suite for 4 and 8 cores. 109

8.1 Memory read requests priority assignment scheme. 136
8.2 Full-system, detailed simulation parameters for the presented criticality-based priority scheme. 140
8.3 Randomly selected 8-core workload sets from SPEC cpu2006 for evaluation of presented scheme. 141

List of Figures

2.1 Capacities and latencies of memory. 11
2.2 Chip multiprocessor (CMP) and memory subsystem architectures. 12
2.3 Typical DRAM *Chip* organization with multiple DRAM *Bank* structures. 13
2.4 A typical memory controller organization. 14

3.1 Simics functional simulator architecture [48]. 21
3.2 GEMS detailed, fully system, microarchitecture simulator overview [49]. 23

4.1 Example of misses and bandwidth MSA histograms of *bzip2* benchmark [16]. 30
4.2 Example of operation of the MSA-based, write-back bandwidth profiling mechanism. 31
4.3 Representative examples of applications from SPEC CPU2006 showing patterns of the memory bandwidth use versus the number of cache ways allocated to them. 34
4.4 Sensitivity analysis of the *Absolute* and *Effective* accuracy of the MSA-based profiler mechanism. 37

5.1 Baseline CMP architecture using a DNUCA last-level cache system that is assumed for the Bank-aware scheme. 44
5.2 Last-level cache banks aggregation schemes. 48
5.3 Example of a CMP cache capacity partitioning using the "Bank-aware" scheme. 51
5.4 Flow chart of "Bank-aware" cache capacity allocation algorithm. 53
5.5 Relative miss rate compared to the fixed-share for *Unrestricted* scheme. . 56
5.6 Relative miss rate of 8-core sets compared to the no-partitioning scheme. 58
5.7 Relative CPI of 8-core sets compared to the no-partitioning scheme. . . . 58

6.1 Baseline multi-chip CMP system assumed for the study of the "Bandwidth-aware" scheme. 64
6.2 Example of *Memory Bandwidth Over-commit* algorithm for a four chips system. 71

6.3 Presented“Bandwidth-aware” resource management framework. 72
6.4 Relative miss rate improvement of UCP+ and proposed Bandwidth-aware schemes. 76
6.5 Relative memory bandwidth use improvements of UCP+ and proposed Bandwidth-aware schemes for different chip configurations over Static-even partitioning scheme. 77
6.6 Detailed simulation results for UCP+ and presented Bandwidth-aware schemes. 79
6.7 Relative IPC improvement of homogeneous multithreaded workloads over static-even partitions on each chip for 32/48/64 concurrent executing thread. 82
6.8 Relative IPC and bandwidth use improvements of heterogeneous multi-threaded workloads over static-even partitions on each chip for 32/48/64 concurrent executing threads. 84

7.1 Three representative categories of miss-rate sensitivity of applications to cache space. 93
7.2 Overall MCFQ scheme on a typical CMP system and an example of allocation on a 16-way cache. 105
7.3 Evaluation of MLP-aware assignment of cache capacity assuming an isolated cache partitioning scheme like UCP [63] on a 4core CMP system. 111
7.4 Comparison of throughput improvements of MCFQ, TADIP, PIPP and UCP schemes for 4 and 8 cores over simple, no cache management LRU scheme. 113
7.5 Cache resource use statistics for the 4-core CMP runs. 115
7.6 Comparison of performance fairness improvements of MCFQ, TADIP, PIPP and UCP schemes for 4 and 8 cores over simple, no cache management LRU scheme. 116

8.1 Typical memory controller organization driving DDRx main memory modules. 123
8.2 Analysis of power and bus utilization improvements as a function of the number of DRAM bank accesses per activation. 127
8.3 Typical capacities and access latencies of every level in memory hierarchy. 129
8.4 DRAM Row buffer policy examples. 130
8.5 System address mapping schemes to DRAM addresses - The example system in figure has 2 memory controllers (MC), 2 ranks per DIMM, 2 DIMMs per channel, 8 banks per rank and 64B cache-line block size. . . 132
8.6 Speedup of targeting 2, 4, and 8 sequential page hits, compared to FR-FCFS. 142

8.7 Speedup of PABS, ATLAS, and proposed scheme relative to FR-FCFS. . 144
8.8 Average number of page-hits per page activation for all schemes 145
8.9 Execution fairness improvements compared to FR-FCFS scheme. 147
8.10 DRAM energy improvements relative to FR-FCFS scheme. 148

Chapter 1

Introduction

While semiconductor technology is moving deeper in the nanometer era, it is well understood now that even though technology scaling will be able to continue providing transistor density improvements, power density and performance improvements will significantly slow. Such frequency and power "walls" of silicon technology scaling have been broadly discussed in recent literature [5, 21, 44, 85]. To leverage the available transistor density, processor designers have refocused their efforts on chip-level throughput rather than targeting single-core performance, by shifting to Chip-Multiprocessor (CMP) architectures that pack increasing numbers of cores and threads on a single chip. During the last decade, industry moved from single-core designs [13] (for the high-performance server processors) to implementations with multiple cores and execution threads.

Over the past two decades, processor speeds have increased at a much faster rate than memory speeds as DRAM (Dynamic Random Access Memory) manufacturers focus on device density and therefore trading speed for DRAM capacity. As a result of this speed difference, the number of processor cycles it takes to access main memory has also increased moving such latency to well over 200-300 cycles and this trend is predicted to continue increasing in the future [31, 83]. Such disparity between processor speed and memory speed is commonly referred as "Memory Wall" in literature [85]. As main memory access is one of the most important limiter in system performance, designers have broadly used many variations of cache designs to reduce the number of memory accesses. Cache misses at the last, higher level of a processor's cache design can potentially stall

a processor for hundreds of CPU cycles in order for the memory subsystem to fetch the required missing data from the slower main memory. Consequently, reducing cache misses was always a key component to sustain high performance and system efficiency and this trend is expected to remain in future CMP designs.

Looking at the processor/memory interconnection, as CMPs have become the norm in high performance system designs, the communication between processor and main memory has been particularly challenged. Technology scaling provides roughly 2x the number of transistors per generation, so when core or thread counts more than double per generation, the result is generally a decrease in the available cache size per core/thread [36]. Such reduction on on-chip cache size in general results in higher miss rates and higher memory bandwidth demands. In addition, designs targeting at overall system throughput, interchange cache size with processing units; leading to smaller cache capacity available per thread. These many-core architectures struggle not only to provide sufficient main memory bandwidth per core/thread, but also to achieve high memory bus utilization efficiency. To make the problem even worse, as typical server processor designs include a small number of memory controllers, a single controller is accepting memory requests from multiple computation streams; destroying any memory access locality that could exists. All these reasons lead to inefficient memory requests scheduling that causes performance reductions and consumes unnecessary energy.

1.1 Summary of Challenges and Solutions

Overall, due to a) the increasing memory latency in processor cycles, b) decreasing cache capacity size available to each core/thread, and c) increasing bandwidth demands to the main memory, the importance of memory-subsystem resource management has become more critical than ever before. Although main memory capacity is managed by OS, most of the other memory resources, like main memory bandwidth, cache capacity

and cache bandwidth, are typically managed by conventional greedy sharing policies that are no longer appropriate. Under high capacity pressure, conventional resource sharing policies lead to destructive interference [6], unfairness [38] and eventually lack of Quality-of-Service (QoS) [14, 38], i.e., lacking the ability to guarantee a certain performance or fairness level.

As a solution, this dissertation investigates the methodology of partitioning the memory-subsystem resources between multiple concurrently executing applications. The philosophy behind the resource partitioning concept is that each resource should be allocated and/or shared between the applications that can most efficiently use such resource. Therefore, the allocation should first take place between the applications that benefit more from the resources (in terms of final system performance and energy efficiency) rather than to the applications that have a higher demand for them. To do so, this dissertation focuses on controlling the contention on the shared *Last-level Cache* (LLC) capacity and the *main memory bandwidth*, as the two most important resources that significantly affect system performance and energy consumption. Overall, to implement an efficient memory-subsystem resource management one will need: a) mechanisms that can dynamically capture the resource requirements for each application for every different shared resource (addition to resource requirements defined by the software), b) microarchitecture-level mechanisms that can enforce specific shared resource allocations, and finally, c) resource allocation algorithms and/or policies that will manage the resource allocation mechanisms based on the applications' requirements. This dissertation presents efficient solutions to all of these three requirements.

Even though cache capacity partitioning schemes have been previously analyzed [7, 63, 72, 76], their study was made assuming either simplified cache hierarchies with no realistic restrictions or complex distributed cache schemes that are difficult to integrate in a real design. As wire delay is gradually becoming the most important design

factor in cache architectures, designers have successfully used banking techniques [19][78] to mitigate the effects of increasing wire delays for short distances. Banked architectures are now the typical design direction for caches in both industry and academia. A typical approach used in academia is the Non-Uniform Cache Architecture (NUCA) designs [37] that is based on assuming non-uniform access latencies to all cache banks of a large L2 cache. As new CMP designs include more cores and cache capacity, a banked L2 cache design is a promising solution that can scale with the number of cores and is able to alleviate wire delay problems. To this end, this dissertation presents a *Bank-aware* partitioning strategy for the CMP-DNUCA architecture, that is able to achieve comparable performance improvements with previously proposed schemes while being consistent with the current industry memory hierarchy trends that is aware of the banking structure of the L2 cache.

Previously proposed cache management schemes suffer from inefficient cache capacity utilization, by either focusing on improving the absolute number of cache misses or by allocating cache capacity without taking into consideration the applications' memory sharing characteristics. Reduction of the overall number of misses does not always correlate with higher performance, as Memory-level Parallelism (MLP) can hide the latency penalty of a significant number of misses in out-of-order execution. This dissertation demonstrates that just targeting the reduction of absolute number of misses introduces significant inefficiencies. A significant percentage of cache space can be allocated to applications with high cache demand rate that cannot actually extract any benefit from the dedicated capacity; while low MLP, miss-latency sensitive workloads with small demand rates can suffocate in small cache partitions. With the number of cores per die is increasing, such wasteful cache management schemes will severely affect performance. As an extension of simple cache partitioning schemes with isolated partitions, this dissertation describes an efficient quasi-partitioning scheme for last-level

caches, named MCFQ, that combines the memory-level parallelism, cache friendliness and interference sensitivity of competing applications, to efficiently manage the shared cache capacity. It is named quasi-partitioning because it mimics the operation of capacity partitioning schemes without actually enforcing the use of isolated partition; while at the same time provides the benefits of pseudo-partitioned schemes. The presented scheme improves both system throughput and performance fairness – outperforming previous schemes that are oblivious to applications memory behavior.

As CMPs are widely deployed in large server systems, these large systems typically utilize virtualization where many independent small and/or low utilization servers are consolidated [65]. Under such environment the resource sharing problem is extended from the chip-level to the system-level. Consequently, to design the most effective future systems for such computing resources, effective resource management policies are critical not only in mitigating chip-level contention, but also in improving system-wide performance and fairness. Limiting optimizations to a single chip can only produce sub-optimal solutions. While previously proposed schemes focus on resource sharing within a chip, this dissertation explores additional possibilities both inside and outside a single chip by proposing a dynamic memory-subsystem resource management scheme that considers both cache capacity and memory bandwidth contention in large multi-chip CMP systems. The described in this dissertation approach uses low overhead, non-invasive resource profilers to project each core's resource requirements and guide the cache partitioning algorithms. The bandwidth-aware algorithm seeks for throughput optimizations among multiple chips by migrating workloads from the most resource-overcommitted chips to the ones with more available resources. The scheme is able to achieve significant reductions in the overall system memory bandwidth use along with reductions in last-level cache miss rates, compared to existing resource management schemes, with a marginal hardware overhead over previous schemes.

Finally, this dissertation describes a solution to efficiently managing memory subsystem resources by coordinating the memory requests, initiated from the cores, with the memory controller policy and the DRAM page-mode operation. Previous proposals to improve efficiency and fairness of main memory employ reordering of requests in the memory controller as the primary control point. A well designed system will saturate the memory interface on a reasonable subset of workloads [67]. However, this leaves an important fraction of workloads where the bandwidth usage is significant, yet not saturated. This dissertation will show that the fullness of the read input queues of the memory controller is relatively low for many workload combinations. Due to the low fullness of the memory structures in the average execution cases, previous policies are inherently ineffective for workloads where the memory interface is below saturation.

As an improvement to previous memory controller policies, this dissertation presents a scheme that is based on two important observations. First, main memory page mode gains, such as power and scheduling conflict reduction, can be realized with a relatively small number of page accesses for each activation. Based on this insight, the DRAM address mapping scheme can be modified to target this small number of hits, enabling more uniform bank utilization and preventing thread starvation in cases of conflict. Secondly, page mode hits exploit spatial reference locality, of which the majority can be captured in modern prefetch engines. Therefore, the prefetch engine can be used to explicitly direct the page-mode operations in the memory scheduler and the priority scheme in the memory controller. To do so, the memory controller page-mode is controlled with prefetch meta-data to enforce a specific number of page mode hits. Finally, the memory request priority scheme includes an intuitive criticality-based scheme where demand read and prefetch operations are ranked based on the latency sensitivity of each operation. Overall such scheme is effective in concurrently improving throughput and fairness across a wide range of memory utilization levels and is particularly effective,

compared to prior work, in improving workload combinations that contain streaming memory references.

1.2 Thesis Statement

Conventional greedy sharing policies are no longer appropriate for managing critical Chip-Multiprocessor (CMP) resources as high resource demand pressure leads to destructive interference and unfairness. Resource partitioning techniques along with careful coordination of resources use between memory hierarchy boundaries, are able to provide significant improvements to the memory management subsystem, enhancing system performance and resource use efficiency throughout the whole hardware stack.

1.3 Contributions

This dissertation makes the following contributions:

1. Presents low overhead, non-invasive, hardware profiler mechanisms based on Mattson's stack distance algorithm (MSA) [50] that can effectively project applications' memory resource requirements and memory sharing behavior for a CMP system. The described profilers are important components for the presented schemes throughout the whole dissertation and can effectively guide the proposed dynamic resource management schemes, allowing them to make resource requirements projections with a very small hardware overhead.

2. It describes a dynamic partitioning scheme for the CMP-DNUCA architecture, consistent with the current industry trends, that is aware of the banking structure of the L2 cache. Results for an 8-core system show that our proposed scheme provides on average a 70% reduction in misses compared to non-partitioned shared caches,

and a 25% misses reduction compared to static equally partitioned (private) caches.

3. Presents a system-wide optimization of memory-subsystem resource allocation and job scheduling. The scheme aims to achieve overall system throughput optimization by identifying over-utilized chips, in terms of memory bandwidth and/or cache capacity requirements. For those chips, a set of job migrations is able to balance the utilization of resources across the whole platform leading to improved throughput. Such a bandwidth-aware scheme is able to achieve a reduction of 18% reduction in memory bandwidth along with a 7.9% reduction in miss rate and an average 8.5% increase in IPC, compared to single chip optimization policies.

4. This dissertation describes a Quasi-partitioning scheme for last-level caches, MCFQ, that combines the memory-level parallelism (MLP), cache friendliness and cache interference sensitivity of competing applications, to efficiently manage the shared cache capacity. The presented scheme improves both system throughput and performance fairness – outperforming previous schemes that are oblivious to applications' memory behavior. The schemes can achieve an average improvement of 10% in throughput and 9% in fairness over the next best scheme on a 4-core CMP, with some specific cases reaching an improvement of 18% and 23% in throughput and fairness, respectively.

5. Finally, the dissertation presents a criticality-based, memory controller requests priority scheme where demand read and prefetch operations are ranked based on the latency sensitivity of each operation. The scheme works in coordination with the page-mode policy and the prefetcher. To do so, memory controller page-mode is directed with prefetch meta-data to enforce a specific number of page mode hits. In parallel, as the scheme solves fairness through the address mapping scheme, memory controller's priority policy is based on the current MLP and other

metrics available within the prefetch engine, for each individual memory request. The scheme demonstrated average gains of 6-7% in throughput over the best prior proposals for medium and high memory utilization levels, in conjunction with improved fairness. Finally, it was found particularly effective in improving workload combinations that contain streaming memory references (up to 30% improvements in throughput and 15% in fairness).

1.4 Dissertation Organization

The rest of this dissertation is organized as follows. Background terminology and related work is discussed in Chapter 2. Chapter 3 summarizes the simulation infrastructure and tool-set that was used throughout the whole dissertation. Chapter 4 describes the hardware-based profilers for applications memory-subsystem resource requirements. Chapter 5 analyses a realistic and cost effective bank-aware cache partitioning scheme for multicore architectures. A bandwidth-aware resource management of the memory subsystem for large CMP systems is presented in Chapter 6. Chapter 7 discusses an efficient approach to quasi-partition large last-level caches, while, chapter 8 describes a cost-effective way to implement a criticality-based, memory controller's priority scheme. Finally, Chapter 9 provides a summary of the dissertation and directions for future work.

Chapter 2

Background Terminology and Related Work

Computer architecture community has heavily studied the topic of memory resource management for the past several years in an effort to minimize the impact of memory and power '"walls" to future architectures. As a result, there are many solutions for improving memory efficiency both in single core and multi-core designs. This dissertation mainly focuses on solving the inefficiencies introduced by CMP architectures rather than improving single thread performance. This chapter first provides a short introduction to the terminology used in caches, main memory and memory controller designs, followed by the related work on memory subsystem resource management. Additionally, related work for the specific problems studied in this dissertation is discussed in detail in the corresponding chapters in order to provide to the reader a qualitative and quantitative comparison with the proposed techniques.

2.1 Memory Subsystem Background

2.1.1 Memory Caches

Caches are one of the most fundamental components in computer architecture. Their main purpose is to bridge the speed difference between processor and memory technology. In an essence, the cache is a smaller, faster memory implemented close to the core which stores copies of the data from the most frequently used main memory locations. The more memory accesses are served by the cache, the faster the average memory access time will be for the system. Therefore, a cache "miss" can significantly affect system

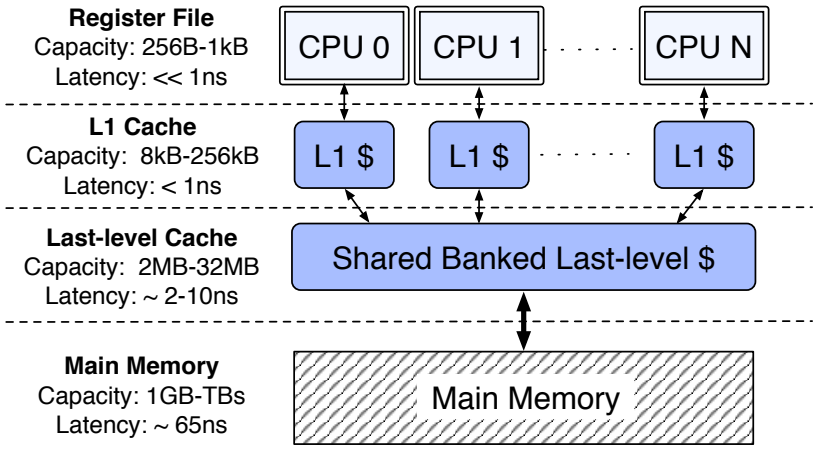


Figure 2.1: Capacities and latencies of memory.

performance. Caches organization and their efficiency/performance is a function of the three basic parameters of the cache [71]: *size*, *associativity*, and *cache-line size* (also known as cache block size).

Cache misses can be classified in three basic categories [82] (known as the *3-Cs*): *compulsory misses*, *conflict misses* and *capacity misses*. Compulsory misses (also known as *cold* misses) are those misses caused by the first reference to a cache-block. Cache size and associativity make no difference to the number of compulsory misses but prefetching and larger cache block sizes can help. Capacity misses are those misses that occur regardless of associativity or block size, solely due to the finite size of the cache and depend on the memory working size of the applications. Finally, conflict misses are the misses caused by the organization of the cache and the cache-block replacement policy used.

With the introduction of CMP designs, the *3-Cs* model is now extended with an additional category of misses, the *memory coherency misses*. These misses are caused by

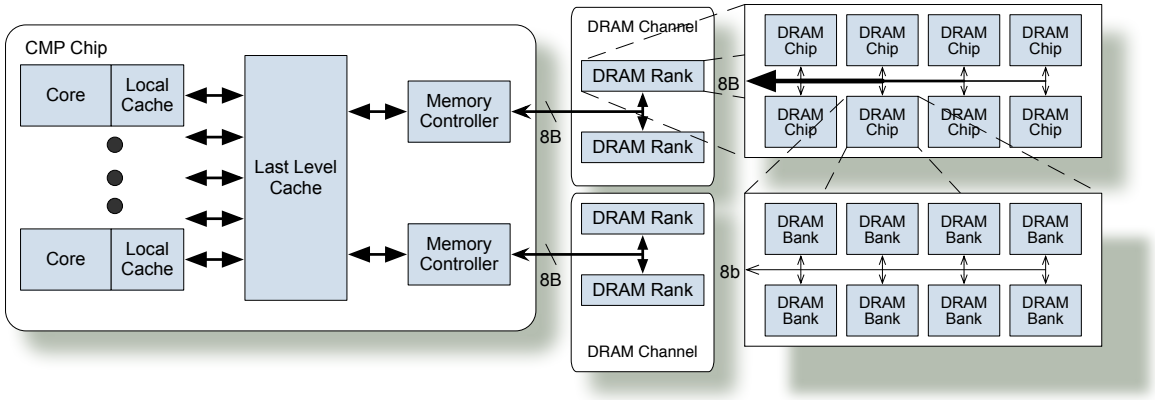


Figure 2.2: Chip multiprocessor (CMP) and memory subsystem architectures.

coherency misses in multithreaded execution when a core that had locally fetched a cache line in its L1 cache, features a new miss to access the same data because another core sharing the same cache line requested an exclusive access to it, invalidating through the coherence protocol all the other local copies of the line in the system.

As contemporary CMP processor designs have evolved to impressive systems on a chip, many high performance processors (eight in current leading edge designs) are backed by large last-level caches containing up to 32 MB of capacity [30]. A typical memory hierarchy that includes a shared, banked, last-level cache is shown in Figure 2.1.

This dissertation will focus on improving cache efficiency with modifications in the cache management system. The presented modifications mainly affect the cache block insertion and replacement policies. In addition, the proper management of cache capacity among the multiple executing threads allows the better use of the cache by the applications that provide the bigger gains in overall system performance and not just single thread improvements. Consequently, the performance and energy improvements are mainly coming from reducing conflict and capacity misses.

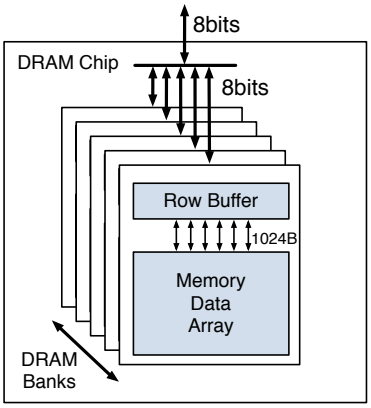


Figure 2.3: Typical DRAM *Chip* organization with multiple DRAM *Bank* structures.

2.1.2 Main Memory

A typical organization of a CMP and memory subsystem is illustrated in Figure 2.2. To maximize memory bandwidth and memory capacity, server processors have multiple *memory channels* per chip. As Figure 2.2 shows, each channel is connected to one or more *DIMMs* (memory cards), each containing numerous *DRAM chips*. These DRAM chips are arranged logically into one or more *ranks*. Within a rank, each DRAM chip provides just 4-8 bits of data per data cycle, and a rank of 8-16 DRAM chips works in unison to produce eight bytes per data cycle. The DRAM *burst-length* (BL) specifies an automated number of data beats that are sent out in response to a single command, commonly 8 data beats, to provide 64Bytes of data. From the time of applying an address to the DRAM chips, it takes about 24ns (96 processor clocks at 4GHz) for the first cycle of data, but subsequent data appear at high frequency, closer to 2-3 processor clocks.

While DRAM devices output only 16-64 bits per request (depending on the DRAM type and burst settings), internally, the devices operate on much larger, 1KB pages (also referred to as *rows*). As shown in Figure 2.3, each DRAM array access causes all 1KB of

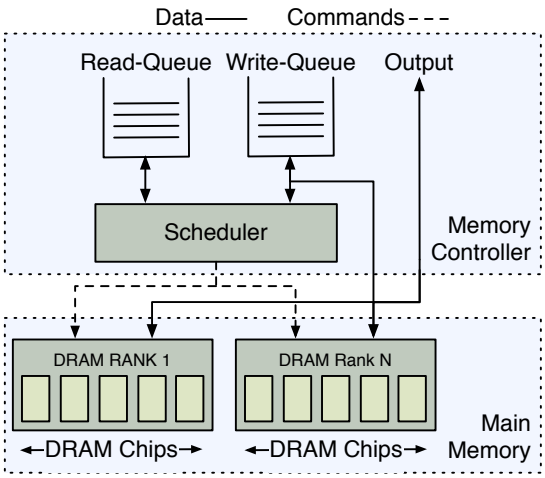


Figure 2.4: A typical memory controller organization.

a page to be read into an internal array called *Row Buffer*, followed by a "*column*" access to the requested sub-block of data. Since the read latency and power overhead of the DRAM cell array access have already been paid, accessing multiple columns of that page decreases both the latency and power of subsequent accesses. These successive accesses are said to be performed in *page mode* and the memory requests that are serviced by an already opened page loaded in the row buffer are characterized as *page hits*.

2.1.3 Memory Controller

A typical memory controller organization is shown in Figure 2.4. DRAM chips are optimized for cost, meaning that technology, cell, array, and periphery decisions are made with a high priority on bit-density. This results in devices and circuits which are slower than standard logic, and chips that are more sensitive to noise and voltage drops. A complex set of timing constraints has been developed to mitigate each of these factors for standardized DRAMs, such as outlined in the JEDEC DDR3 standard [28]. These timing constraints result in "dead times" before and after each random access; the processor

memory controller's job is to hide these performance-limiting gaps through exploitation of parallelism. This dissertation proposes optimizations to improve the access latency and increase the sustained efficiency of the IO interface connecting main memory with the CMP. This is accomplish through policy improvements to the memory scheduler and how operations are mapped to the DRAM devices.

2.2 Related Work in Cache Management

2.2.1 Isolated Cache Partitions

To solve the problem of destructive interference in the last-level-caches, researchers have proposed to partition the cache among different threads [7, 63, 72, 76]. This is commonly done by means of *way-partitioning* i.e, in a set-associative cache each application is given the exclusive ownership of a fixed number of cache-ways. Suh *et al.* [76] proposed a greedy heuristic to allocate cache-ways proportionally to the incremental benefit that the thread gets from that allocation. Later on, Qureshi *et al.* [63] proposed Utility Based Cache Partitioning (UCP). They used utility monitors to estimate the utility of assigning additional ways to a thread and allocate ways based on this utility. This dissertation presents "Bank-aware" proposal applying the basic concept of UCP on realistic CMP implementations with DNUCA-like caches to provide a scaling solution [33]. In the "Bank-aware" partitioning scheme there is no DNUCA cache and therefore UCP and "Bank-aware" proposal are identical. This dissertation also extends Qureshi's work with a "Bandwidth-aware" resource managing scheme that is taking in to consideration both cache capacity and memory bandwidth contention in large multi-chip CMP systems [32]. Moreto *et al.* [52] describes a partitioning scheme that uses the MLP information and it is based on the same principles with our approach.

2.2.2 Dead-time Management

Cache lines with poor temporal locality occupy valuable cache resources without providing any actual benefits (cache hits). To minimize the life time of these lines, Qureshi *et al.et al.* proposed Dynamic Insertion Policy (DIP) [61]. DIP identifies dead lines and inserts them at LRU position instead of MRU position resulting in their quick eviction. This allows more useful lines to be retained in the cache enabling better utilization of capacity. A subsequent proposal by Jaleel *et al.* [26] extends DIP to manage dead-time in the multi-core environments. They proposed Thread Aware Dynamic Insertion Policy (TADIP) which can adapt to memory requirements of competing applications.

2.2.3 Cache Pseudo-Partitioning

A recent proposal *Promotion/Insertion and Pseudo-Partitioning* (PIPP) [86] combines the ideas presented in UCP and TADIP to support a cache pseudo-partitions scheme. PIPP can provide good isolation for highly reused lines but due to their insertion policy, still suffers from high destructive interference in the lower parts of the LRU stack. Rafique *et al.* [64] proposed an OS controlled technique where it is possible for applications to steal lines from other applications if they are not using their allocations effectively. This dissertation presents a "MLP and Cache Friendliness aware Quasi-partitioning" scheme (MCFQ) that implements quasi-partitioning by assigning different insertion points to applications keeping in view their MLP, cache memory behavior (Friendly, Fitting, Thrashing) and their interference sensitivity. This allows MCFQ to efficiently utilize the capacity while reducing the effects of interference. Herrero *et al.* in [17] proposed "Elastic Cooperative Caching" for NUCA caches. This scheme detects different cache requirements of applications and distributes cache resources accordingly, allowing the creation LLC partitions that can be private, shared or both based on the demands of applications.

2.2.4 QoS/Fairness

Apart from the throughput driven techniques, researchers have proposed cache capacity partition algorithms that focus on improving fairness and/or Quality of Service (QoS) [14, 20, 23, 38]. Kim *et al.* [38] highlighted the importance of enforcing fairness in CMP caches and proposed a set of fairness metrics to evaluate fairness optimizations. Chang *et al.* [7] proposed time-sharing of cache partitions, which transforms the problem of fairness to a problem of scheduling in a time-sharing system. Iyer *et al.* [23] proposed a framework for enforcing QoS characteristics in a system based on a trial-and-error scheme to fit the QoS targets. That work was later on extended by Zhao *et al.* [88] with a set of counters, named CacheScouts, that monitored their QoS characteristics in a system and, based on them, made resource management decisions. Researchers have used the contention characteristics of applications [29, 90] to make applications' co-scheduling decisions.

2.2.5 Cache Replacement Policies

Qureshi *et al.* [62] have proposed an MLP-aware cache replacement policy. They utilize MLP-cost in addition to recency information when finding victims for replacement. Recently, Jaleel *et al.* have proposed a replacement policy based on Re-Reference Interval prediction (RRIP) [27]. Their scheme predicts the interval after which each block in the set is likely to be accessed and prevent the blocks with distant re-reference interval to evict a block that is predicted to have a re-reference in the near future. Although these replacement policies help in improving cache performance, they do not provide any direct control over the number of cache lines each thread can maintain in the cache and thus destructive interference is still present. A cache-replacement policy can provide some benefits of cache partitioning schemes with proper handling of eviction and allocation of lines but none of the above schemes have such control compared to our proposed scheme.

2.3 Related Work in Memory Controller Policies

Rixner *et al.* [66] first described the *First-Ready First-Come-First-Serve* (FR-FCFS) scheduling policy that prioritizes row-hit requests over other requests in the memory controller queue. This proposal utilizes a combination of a column centric DRAM mapping scheme, similar to the one in Figure 8.5(a), combined with FR-FCFS policy. Such approach though create starvation and throughput deficiencies when applied to multi-threaded systems as described by Moscibroda *et al.* [53]. Prior work attempts to mitigate these problems through memory requests scheduling priority. Mutlu *et al.* based their "Stall Time Fair Memory" (STFM) scheduler [55] on the observation that giving priority to requests with opened pages can lead to significant introduction unfairness in the system. As a solution they proposed a scheme that identifies threads that are stalled for a significant amount of time and prioritize them over requests to open-pages. On the average case, STFM will operate similarly to FR-FCFS mapping.

The *Adaptive per-Thread Least-Attained-Service* memory scheduler (ATLAS) [39] proposal tracks attained service over longer intervals of time. Following the same logic, *Parallelism-aware Batcher Scheduler* (PA-BS) [56] ranks lower the applications with larger overall number of requests stored in every "batch" formed in the memory queue. Since streaming workloads inherently have on average a large number of requests in the memory queue, they are scheduled with lower priority. A recent work, *Thread Cluster Memory Scheduler* (TCM) [40] extends the general concept of the ATLAS approach. In TCM, unfriendly workloads with high row-buffer locality, that utilize a single DRAM bank for an extended period of time, are given less priority in the system, such that they interfere less frequently with the other workloads.

Lin *et al.* [45] proposed a memory hierarchy that coordinated the operation of the existing prefetch engine with the memory controller policy to improve bus utilization and throughput. In their hierarchy, the prefetch engine issues requests that are spatially close

to recent demand misses in L2 with the memory controller sending the requests to memory only when the memory bus is idle. Their prefetcher relies on a column-centric address hash which introduces unfairness in the system that is not addressed in the proposal. Following that, Lee *et al.* [43] propose a *Prefetch-Aware* controller priority, where processors with a history of wasted prefetch requests are given lower priority. Finally, "Micro-pages" proposal from Sudan *et al.* [75] describe a scheme that uses smaller than typical OS page sizes in an effort to co-locate multiple frequently used pages in the same DRAM row buffer. Their solutions includes software and hardware migration mechanisms to move data in such small pages. This approach requires significant software effort to achieve a greater number of page hits which we found to be unnecessary to realize page-mode benefits.

Chapter 3

Simulation Methodology, Tools and Workloads

Even though rapid prototyping techniques and EDA (Electronic Design Automation) tools have drastically evolved during the last decade, microarchitecture simulation is still one of the most important techniques used by computer architects to evaluate their research ideas and proposals. The challenges in achieving correct and accurate simulation results are twofold: a) the target machine has to be simulated with sufficient details (that can be extended up to cycle-accurate and performance validated simulators), and b) such target machine has to driven with a realistic and representative workload.

This dissertation uses a combination of a full-system timing/functional simulator SIMICS [48], along with a detailed microarchitecture simulator GEMS [49], to evaluate the proposed memory subsystem resource management schemes. Even though each proposed scheme in this dissertation requires its own modifications in the original tools, various system configurations and addition of profiling mechanisms, the basic evaluation methodology and tool-chain remains the same. For the majority of the dissertation SPEC CPU2006 [16] was used as a representative, general purpose, benchmark suite in order to create single thread and multiprogram workloads sets. In addition, to evaluate the multithreaded capabilities, SPLASH-2 [84], Apache [11] and SPECjbb2005 [60] benchmarks suites were used.

The remaining of this chapter includes a brief introduction to each tool followed by a description of the workload suites used to drive the evaluation process. As for the case of the related work, each proposed scheme includes a separate description of the evaluation

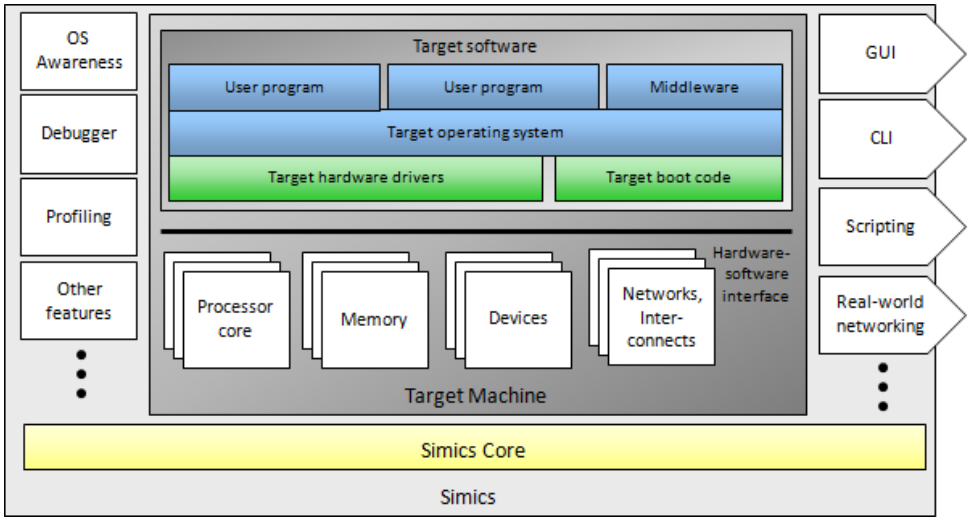


Figure 3.1: Simics functional simulator architecture [48].

methodology and simulator modification/addition to enable the comparison against prior proposals.

3.1 Full System Timing Simulator

This dissertation uses *Simics* from Virtutech [48] as a full system timing simulator. *Simics* is a very powerful simulator that is able to boot and run unmodified operating systems and their applications on the target simulated machine. Effectively, Simics is a system-level instruction set (ISA) simulator that is able to accurately simulate the functionality of each ISA instruction in the system. The basic functionality of Simics is unaware of the exact timing to complete a single event and considers the execution of an entire instruction, an exception or an interrupt as an atomic operation in the simulation process that it takes exactly one cycle. The basic configuration used in this dissertation (Simics 3.0) simulates an existing SPARC-v9 SMP processor, the UltraSPARC III+, along

with the necessary memory modules, motherboard chipset, network card and hard disk to allow booting and running a full, unmodified version of Solaris 10 operating system using an SMP kernel. Simics is based on a simplified, in-order, 5 stages pipeline along with atomic memory operations that take a single cycle to complete. Such configuration restrictions can be alleviated with the use of additional modules that can be hooked up with the appropriate API, allowing the simulation of out-of-order, aggressive superscalar processors along with realistic multi-level memory hierarchies. An overview of the Simic's architecture is illustrated in Fig. 3.1. In essence, in this dissertation's evaluation methodology Simics provides the necessary infrastructure to simulate a full, real machine at the functional level while the detailed cycle-accurate, timing simulation of a realistic processor and memory hierarchy is left to GEMS as an external module added to Simics API hooks.

3.2 Detailed Microarchitecture Simulator

As a detailed microarchitecture simulator this dissertation uses the multifacets general execution-driven multiprocessor simulator (GEMS) toolset from the University of Wisconsin [49]. As described in the previous section, Gems in combined with Simics to provide a detailed, cycle-accurate simulator by decoupling simulation functionality and timing. Simics provides a robust environment to boot an unmodified OS along with the functional simulator. Gems timing modules interact with Simics to determine when Simics should execute an instruction. However, what the result of the execution of the instruction is ultimately dependent on Simics. Therefore, the two tools operate in a lock-step mode. Even though, GEMS decouples functional simulation and timing simulation, the functional simulator is still affected by the timing simulator, allowing the system to capture timing-dependent effects.

The basic architecture of GEMS is illustrated in Fig. 3.2. Its basic functionality

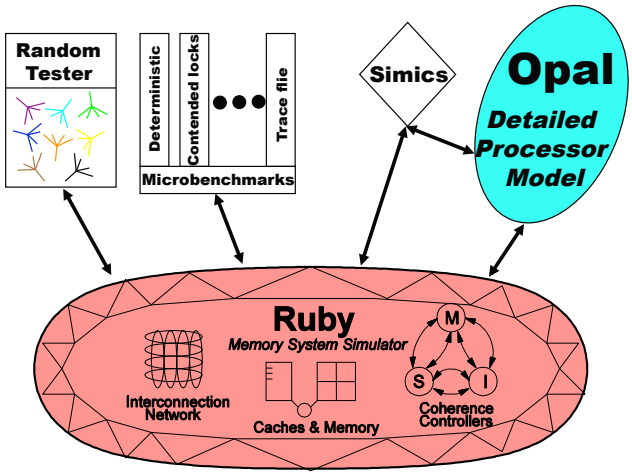


Figure 3.2: GEMS detailed, fully system, microarchitecture simulator overview [49].

is divided between two basic components: *Ruby* and *Opal*. Ruby is the most important component and is the basic timing simulator of a multiprocessor memory system that models: caches, cache controllers, system interconnect, memory controllers, and banks of main memory. Ruby combines hard-coded timing simulation for components that are largely independent of the cache coherence protocol (e.g., the interconnection network) with the ability to specify the protocol-dependent components (e.g., cache controllers, coherence protocol) in a domain-specific language called SLICC (Specification Language for Implementing Cache Coherence). When Ruby is used stand-alone it simulates a simplified in-order processor for every core in the system. To simulate more advanced cores, the additional *Opal* module has to be used along with *Ruby* in *Simics*. Opal models a SPARC ISA, out-of-order, superscalar, deeply-pipelined processor core. Opal is configured by default to use a two-level gshare branch predictor, MIPS R10000 style register renaming, dynamic instruction issue, multiple execution units, and a load/store queue to allow for out-of-order memory operations and memory bypassing. Because Opal runs ahead of the Simics functional processor, it models all wrong path effects of

instructions that are not eventually retired. Opal implements an aggressive implementation of sequential consistency, allowing memory operations to occur out-of-order and detecting possible memory ordering violations as necessary.

Overall, the combination of Ruby, Opal and Simics allows a very detailed and accurate simulation of almost any CMP and memory-subsystem configuration. Some more advance features that are missing from a typical contemporary core, like cache-line prefetchers and banked DNUCA-like last-level caches were added on top of Ruby. More details about modifications and additions in the baseline simulation tool-set is describe in each individual chapter.

3.3 Benchmark Suites

Throughout the dissertation, a number of benchmarks suites was used to provide the necessary workloads to drive the simulation tools. The suites include either single-thread, scientific and/or typical representative applications that can be used standalone or combined together to form multiprogrammed workload sets, or multithreaded benchmarks that represent typical commercial and scientific multithreaded applications. This section provides a short description of each benchmark suite to help the reader better understand the provided evaluation results.

3.3.1 SPEC CPU2006

To evaluate the performance of a single core configuration or form multiprogrammed workloads set, SPEC cpu2006 [16] was used. SPEC cpu2006 is the most popular set of benchmarks, mainly used in the computer architecture society, that is designed to test the CPU performance of a modern server computer system. The included benchmarks focus their efforts not only on the systems processor but also on the memory architecture

and the implemented/selected compiler since the overall performance of a system is tightly dependent on all of them. The benchmark suit itself consists of 29 programs (12 integer and 17 floating-point applications), each one of them having unique features that enable them to stress various aspects of a modern microprocessor like ALU, branch predictor, L1/L2/L3 caches and memory controller designs.

3.3.2 SPEC JBB2005

SPEC JBB2005 (Java Business Benchmark) [60] is SPEC's benchmark for evaluating performance of Java on the server-side by emulating a three-tier client/server system (with emphasis on the middle tier). SPEC JBB2005 stresses the implementations of the JVM (Java Virtual Machine), garbage collection, computation threads and some aspects of the operating system. The performance measure of SPEC JBB2005 extends to CPUs, caches, memory hierarchy and the scalability of shared memory processors (SMPs). In general, the workload stress the object oriented characteristics and real-world characteristics like the prevalent use of XML processing. SPEC JBB2005 is a self contained benchmark that emulates a 3-tier system, the most common type of server-side Java application today. SPEC JBB2005 can be configured on three distinct dimensions: length of run, warehouse sequence, and garbage collection behavior. One can control the upper and lower bound of the number of warehouses as a sequence along with controlling the sequence increment. The other controllable properties are the length of the run, the ramp up and ramp down times of each JVM and the length of each warehouse run.

3.3.3 SPLASH-2

To analyze the performance efficiency of multithreaded workloads the Stanford ParalleL Applications for SHared memory (SPLASH-2) suite [84] is used in this dissertation. SPLASH-2 is the second version of the initial SPLASH suite and contains

programs that are more representative of computations in scientific, engineering and graphics domains, and are more architecturally aware than the first version. The suite includes parallel applications that target cache coherent shared address space designs. The SPLASH-2 suite consists of a mixture of complete applications and computational kernels. It currently has 8 complete applications and 4 kernels, which represent a variety of computations in scientific, engineering, and graphics computing. A brief description of each application can be found in Woo *et al.* [84].

Each application can create a user defined number of homogeneous threads that work on a different subset of data. In addition, their working set can be scaled to fit the size of the available on-chip and off-chip memory capacity. The suite does not support a specific implementation of threading libraries and barriers and it is up to the user to provide the best implementation that fits to his/her system. In this dissertation, a POSIX threads implementation optimized for Solaris lightweight threads (LWT) infrastructure was used [15]. In addition, the applications and kernels were instrumented with Simics' "Magic" instructions to indicate the beginning and ending of the critical computation kernels. Such instrumentation allows the simulation of only the performance critical execution phase and avoids spending resources simulating the long memory initialization and thread synchronization part of code.

3.3.4 Apache Web-server

Apache HTTP server [11] is one of the most popular commercial open-source web servers software available that is used by millions of World Wide Web (WWW) sites around the world. Apache2 was used as a representative, multi-threaded, commercial workload that is able to scale to a significant number of threads and working set sizes. As Apache2 is a web server that serves HTTP requests from a client, one will need a way to produce a distribution of requests to the server. For this purpose the Surge (Scalable

URL Reference Generator) workload generation tool [1] from Boston University was used. The tool is based on analytical models of web use and generates references matching empirical measurements of severs size distribution, request size distributions, temporal locality of references and idle periods of individual users. Overall, the tool incorporates the behavior of single user by combining a set of distributions characteristics in order to create a representative web workload.

Chapter 4

Profilers for Application's Memory-subsystem Resource Requirements

In order to dynamically profile the memory-subsystem resource requirements of each core in a typical CMP system, this dissertation describes a prediction scheme that is able to estimate both cache misses and memory bandwidth requirements. This prediction scheme is contrasted against typical profiling models in that it estimates resource behavior of all possible configurations concurrently, as opposed to profiling the currently configured resources. The described monitoring schemes are a key component in almost all the proposed techniques in this dissertation and focus on monitoring the shared, last-level cache in CMP systems. To do so, each core has a dedicated *Cache Profiling* circuit that tracks its shared resource (cache capacity and memory bandwidth of last-level cache) requirements. Such profiling circuit is independent of the memory subsystem and is able to non-invasively monitor the behavior of an application running on a core. Each *Cache Profiling* circuit is unaware of the workloads executing on other cores and assumes that the whole cache is available to the monitoring core. A typical use of the scheme in this dissertation is based on the notion of epochs. During an epoch, the profiling circuit constantly monitors the behavior of each core of the overall system. When an epoch ends, the profiled data of each core are passed to the appropriate memory resource management algorithm to find the ideal resource assignments for each one of the cores.

Overall, the prediction model is based on Mattson's stack distance algorithm (MSA), which was initially proposed by Mattson *et al.* [50]. The initial purpose of the

algorithm was to reduce the simulation time of trace-driven caches by determining the miss ratios of all possible cache sizes with a single pass through the trace. The basic idea of the algorithm was later used for efficient trace-driven simulations of a set associative cache [18]. More recently, hardware-based MSA algorithms have been used for CMP system resource management [63][89]. To predict memory access behavior in addition to cache miss rates, the proposed scheme extends previously described hardware solutions. Specifically, it estimates the memory write traffic produced by the eviction of modified lines in a write-back cache. In the following subsections, a description of the baseline MSA algorithm for profiling LLC misses will be presented followed by the description of the additional structures needed to predict the memory bandwidth requirements.

4.1 Last-level Cache Capacity

Mattson's stack distance algorithm (MSA) is based on the inclusion property of the commonly used *Least Recently Used* (LRU) cache replacement policy. Specifically, during any sequence of memory accesses, the content of an N sized cache is a subset of the content of any cache larger than N. To create a profile for a K-way set associative cache $K+1$ counters are needed, named $Counter_1$ to $Counter_{K+1}$. Every time there is an access to the monitored cache we increment only the counter that corresponds to the LRU stack distance where the access took place. Counters from $Counter_1$ to $Counter_K$ correspond to the *Most Recently Used* (MRU) up to the LRU position in the stack distance, respectively. If an access touches an address in a cache block that was in the i-th position of the LRU stack distance, we increment the $Counter_i$ counter. Finally, if the access ends up being a miss, we increment the $Counter_{K+1}$. The *Hit Counter* of Fig. 4.1 demonstrates such a MSA profile for *bzip2* of SPEC CPU2006 suite [16] running on an 8-way associative cache. The application in the example shows a good temporal reuse of stored data in the cache since the MRU positions have a significant percentage of the hits over the LRU one.

The graph of Fig. 4.1 can change accordingly to each application's spatial and temporal locality. Such an MSA-based profiling allows us to monitor each cores cache capacity requirements during the execution of an application and based on which we can find the points of cache allocation that can benefit the miss ratio the most.

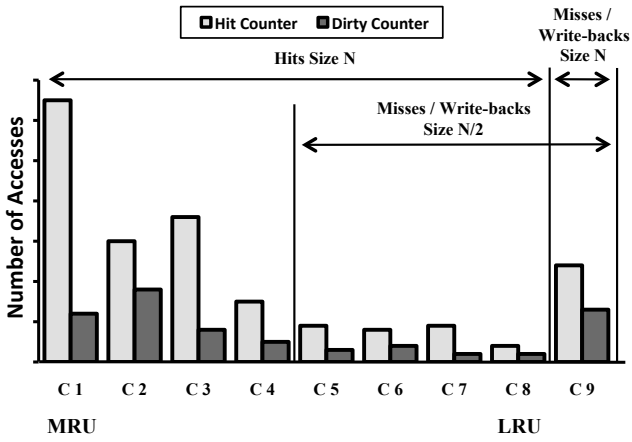


Figure 4.1: Example of misses and bandwidth MSA histograms of *bzip2* benchmark [16].

4.2 Memory Bandwidth

In addition to capacity misses, the MSA circuit was augmented to estimate application's required memory bandwidth. There are two components that must be addressed: *a*) read bandwidth due to *cache fills*, and *b*) write bandwidth due to cache evictions (dirty *write-backs* to memory). The bandwidth needed for fetching data from main memory required by a cache miss (*cache fills*) can be derived from the previously described MSA circuit. The number of misses estimated by MSA is proportional to the memory bandwidth needed to fetch the missed data; with each miss representing a cache-line of bandwidth.

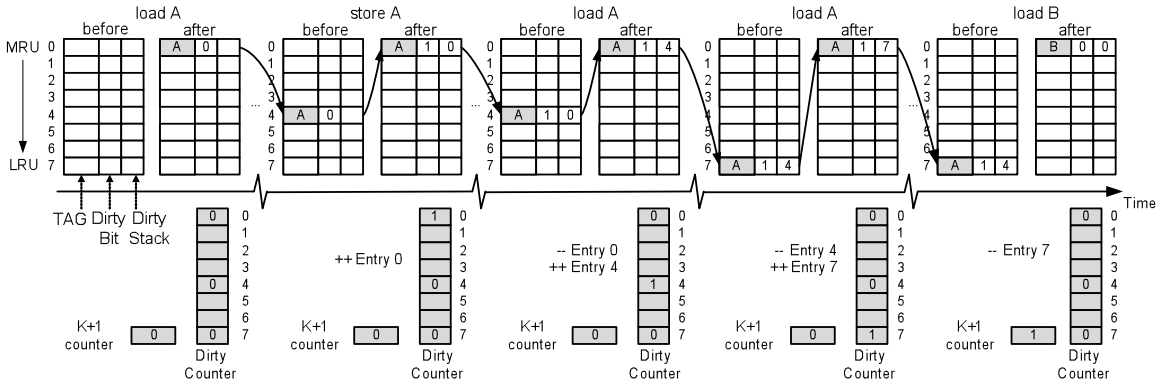


Figure 4.2: Example of operation of the MSA-based, write-back bandwidth profiling mechanism.

To project the *write-back* bandwidth, we exploit the following property. In the MSA structure, hits to dirty lines indicate a write-back operation if the cache capacity allocation is smaller than its stack distance. Essentially, the store data must have been written back to memory and re-fetched since the cache lacked the capacity to hold the dirty data. This process is complicated in that only one write-back per store should be accounted for.

To track these possible write-backs to memory we added a *Dirty Bit* and a *Dirty Stack Distance* indicator (register) for each cache line tracked in the MSA structure. In addition, we add a dirty line access counter, named *Dirty Counter* for each cache way. The *Dirty Stack Distance* is used to track the largest stack distance at which a dirty line has been accessed. We cannot simply update the *DirtyCounter* access counter on each hit of a dirty line, since this will give multiple counts for one store. In addition, we cannot reset the *Dirty Bit* on an access, since a future access to the same line at a greater stack distance must be accounted for in the dirty access counters. Essentially, we must track the greatest stack distance that each store is referenced. Pseudocode 1 describes how we update the *Dirty Bits*, *Dirty Counters* and *Dirty Stack Distance* registers.

Pseudocode 1 Write-back dirty access hardware description.

```
if (access is a hit) {/* Handling Dirty Counters & Dirty Stack Distance */
   hit_distance <= stack distance of hit
   hit_dirty <= dirty bit set in hit entry
   dirty_stack_distance <= dirty distance value in hit entry
   if (hit_dirty and (hit_distance > dirty_stack_distance)) {
         DirtyCounter[dirty_stack_distance] - -;
         DirtyCounter[hit_distance] ++;
         dirty_stack_distance = hit_distance;
   }
} else{ /* access is a miss */
         /* Handle deallocation of evicted line */
         DirtyCounter[dirty_stack_distance] - -;
         K+1_Counter++;
         /* Handle new line allocation */
         dirty_bit = 0
}
/* Handling a store operation */
if (is_store) {
         dirty_bit = 1
         dirty_stack_distance = 0
}
```

The dirty bit is only reset when the line is evicted from the cache. At eviction time an additional counter ($K+1$ counter of Fig. 4.1) tracks the number of write-back operations. This number gives the write-back rate for a cache size corresponding to the maximum capacity tracked. The projection of write-back rates can then be made from the *DirtyCounters*. For each cache size projection, the sum of all counters larger than the allocated size indicates the number of write-back operations sent to main memory.

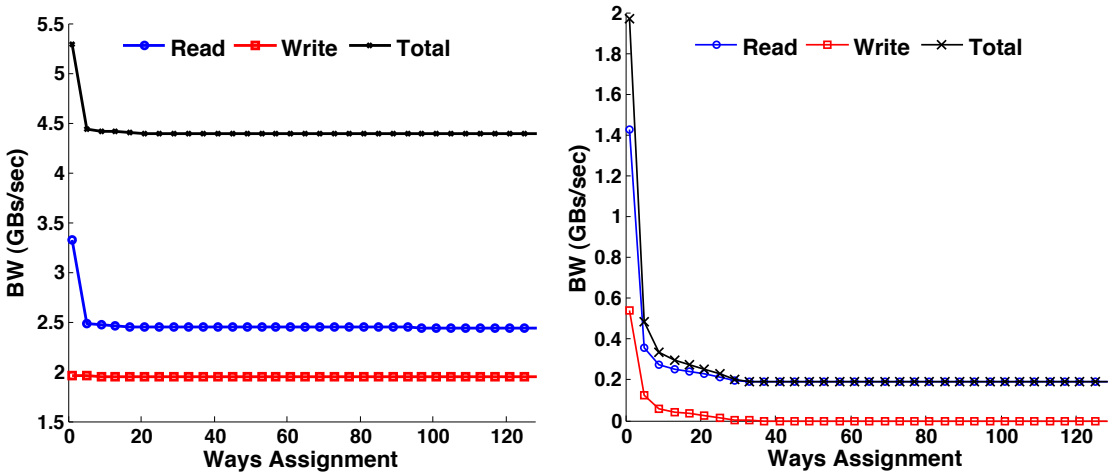
Fig. 4.2 demonstrates how the maximum stack distance at which a given store is referenced is tracked. In the example, we start with a store to a previously allocated clean line. This will set the *Dirty Bit* to 1, set the *Dirty Stack Distance* to 0, and increment the entry '0' of the histogram. Essentially, if the cache had no free capacity, the store

operation would immediately produce a write to memory. Following the store operation, we show two load hits of the MSA structure. The first hit is at a stack distance of 4. As such any cache that is smaller than 4 ways would have not been able to contain the store data. We then move the accounting for the store from entry 0 to entry 4 of the histogram. As we detect a load that hits entry 7, we must move the accounting to entry 7. If the line is evicted, this line results in a write-back for any of the captured sizes. The key insight shown here is that each store will result in exactly one net increment across the histogram and *K+1 Counter* (*K+1 Counter* of Fig. 4.1). Increments to larger stack distances leading to the *K+1 Counter* are more powerful in that avoiding the write-back to memory requires a larger cache assignment.

Overall, the proposed MSA histograms allow making a prediction for both the number of misses and the required memory bandwidth dependent on the number of ways that a core is assigned. An example of such an MSA profile is shown in Fig. 4.1 where both MSA histograms and dirty evictions are shown.

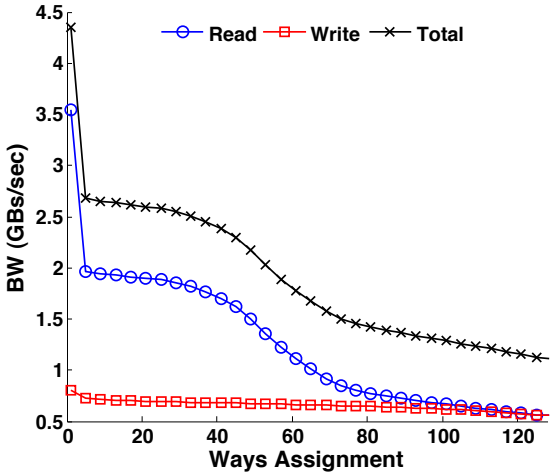
4.3 Examples of Real Applications' Bandwidth Requirements

Fig. 4.3 demonstrates the three most representative categories of memory bandwidth requirements that we observed examining SPEC CPU 2006 [16] suite. As mentioned before, the read bandwidth is highly correlated with cache miss rate and therefore previous techniques indirectly account for it. From the figure, *milc* features a write rate almost equal to the read rate. In this workload, complex matrix data structures are modified at each iteration, producing a high fraction of modified data. In contrast, the *calculix* benchmark uses cache blocking of matrix multiplications and dot product operations to condense and contain store data within the cache. As such, the memory traffic becomes read only beyond the blocking size. As another example of the workload variation we included *gcc*. In the *gcc* workload smaller caches produce a more read dominated pattern, while larger caches



(a) Milc benchmark.

(b) Calculix benchmark.



(c) Gcc benchmark.

Figure 4.3: Representative examples of applications from SPEC CPU2006 showing patterns of the memory bandwidth use versus the number of cache ways allocated to them.

become write dominated. This behavior is due to the code generation aspect. As the cache grows, data tables within the compiler become cache-contained, leaving only the write-back traffic of the generated code.

4.4 MSA-profiler Implementation Overhead and Accuracy

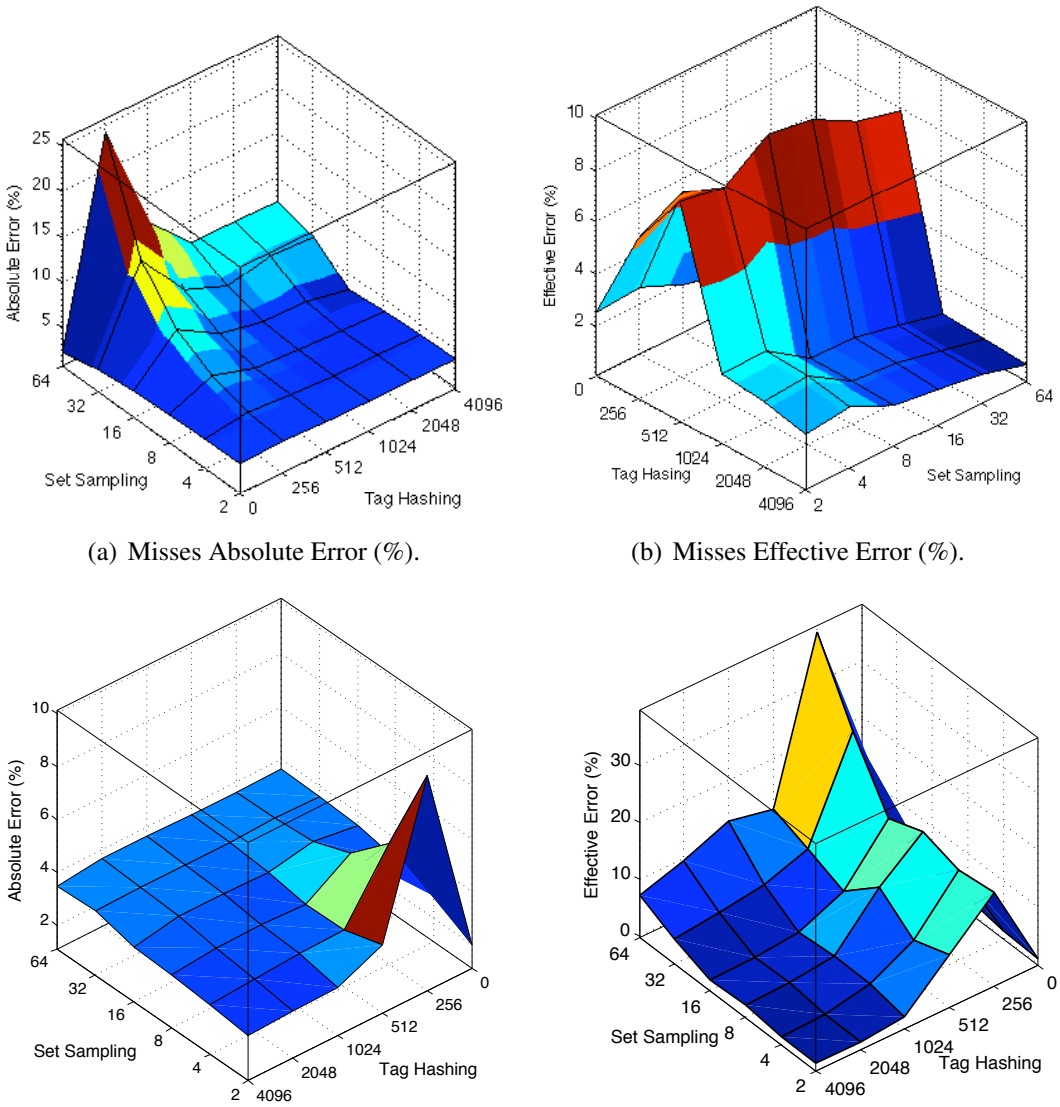
The hardware overhead of the profiling structure is primarily defined by the implementation of the necessary cache directory tag shadow copy. These cache block tags are necessary for identifying which cache block is assigned at each one of the *hit* and *dirty evictions* counter pairs of Fig. 4.1. Additional overhead is introduced by the implementation of the *Hit Counters*, *Dirty Counters* and *Dirty Stack Distance* registers themselves for each cache way, but since those counters are shared over all the available cache-ways, their overhead is significantly lower than the cache block tag information for every set.

A naive implementation would require a complete copy of the cache block tags for each cache set in each one of MSA profilers, which is prohibitively high. The overhead can be greatly reduced using: *a*) *partial hashed tags* [35], *b*) *set sampling* [34], and *c*) *maximum assignable capacity* reduction techniques. With *partial hashed tags* one can use less than full tags to identify the cache blocks assigned at each counter pair thus reducing the storage overhead. Hashing is necessary to reducing the aliasing problem of using less than full tags. *set sampling* involves the profiling of a fraction of the available cache sets and therefore it also reduces the number of stored cache tags in the circuit. In addition, the *maximum assignable capacity* approach assumes that the number of cache-ways that can be assigned to each core is less than the overall number of available cache-ways. In that case, the number of counter pairs are reduced to the maximum number of assignable ways per core. The first two reduction techniques are subject to aliasing, which introduces errors and affects the overall accuracy of the profiling circuit. In addition, the

maximum assignable capacity can potentially restrict the effectiveness of the partitioning scheme by not dedicating bigger portions of a cache to a specific core.

Fig. 4.4 demonstrates the errors introduced by *partial hashed tags* and *set sampling* techniques in the MSA profilers, for both misses and bandwidth, compared to their unrealistic full implementations. The analysis target is to find a configuration that can significantly reduce the necessary overhead to a realistic implementation with an acceptable error rate. The errors estimated as an average error over the analysis of the whole SPEC CPU2006 suite for a slice of 100M instructions per benchmark using the detailed GEMS implementation of the scheme. The full implementation profilers are able to accurately monitor the requirements and do not introduce any errors since they keep all the necessary information from the simulator. We provide two space exploration error analysis, *Absolute error* and *Effective error*. The *Absolute error* represents the aliasing error of the actual raw data in the counters for all configurations. On the other hand, the *Effective error* is estimated over the information that the algorithms use to estimate the ideal cache size. *Set sampling* was selected to change from 1-in-2 up to 1-in-64 samples per cache set and *partial hashed tags* changed from 0 (no partial tags) to 4096 (using only 12 bits of address tags). To mitigate the aliasing problem, the *partial hashed tags* use a randomly created network of XOR gates to hash the partial tags. The XOR tree overhead is very small in comparison to the necessary number of counters and only one copy per MSA-profiler is necessary. There are two rules for choosing a configuration and minimize the hardware overhead: The hardware overhead is a) proportional to the size of *partial hashed tags* , and b) inversely proportional to the *set sampling*. Therefore, we ideally want to keep a small number of tag bits and use a large number of *set sampling*.

From Fig. 4.4 clearly shows that as the *set sampling* number increases so does the error rate. In addition, an increased number of *partial hashed tags* can significantly improve the error rates especially for the case of *Effective error*. For most cases a small



(a) Misses Absolute Error (%).

(b) Misses Effective Error (%).

(c) Bandwidth Absolute Error (%).

(d) Bandwidth Effective Error (%).

Figure 4.4: Sensitivity analysis of the *Absolute* and *Effective* accuracy of the MSA-based profiler mechanism.

Table 4.1: Overhead of the presented MSA-based profiler.

Structure Name	**Overhead Equation**	**Overhead**
Tags Size	$Tag\_width * Cache\_ways * (Sets/Set\_sampling)$	54 kbits
LRU Stack Distance	$((LRU\_pointer * Cache\_ways) + Head/Tail) * (Sets/Set\_sampling)$	27 kbits
Dirty Bits	$Cache\_ways * (Sets/Set\_sampling)$	2.25 kbits
Dirty Counters	$Cache\_ways * Counter\_size$	4.5 kbits
Dirty Stack Distance Registers	$Cache\_ways * Register\_size$	27 kbits
Hit Counters	$Cache\_ways * Counter\_size$	2.25 kbits

number of *partial hashed tags* introduce a significant error which indicated that we need to choose a big number of bits in the tags. Fig. 4.4(a) and Fig. 4.4(c) show that the estimation of misses is more sensitive to the selected configuration that the memory bandwidth use, introducing a variation in the absolute error. On the other hand, Fig. 4.4(b) and Fig. 4.4(d) demonstrate the opposite trend for the effective error. This is a strong indication that the algorithm's decisions are more sensitive to the bandwidth use and we should choose a configuration that favors those decisions more. Furthermore, for high number of partial tags and set sampling, the effective errors are significantly smaller than the absolute errors which allow us to be more elastic on the final selected configuration, improving the overhead.

Taking all the previous trends into consideration, using 11 bit *partial hashed tags* (Tag hashing 2048) combined with 1-in-32 *set sampling* produces an average *Absolute error* rates of 6.4% for misses and 5% for bandwidth. On the other hand, the *Effective error*

was estimated to be 1.3% for misses and 3.6% for bandwidth compared to the profiling accuracy obtained using a full tag implementation. Such error bounds are inline with other set-sampling based monitor schemes like UMON [63] and CacheScouts [88]. The first one concludes that 1-in-32 *set-sampling* is enough for their profiler and the latter reports error rates of 6% for their 1-in-128 set sampling of cache occupancy for scientific workloads. To further improve overhead, the *Intra-chip partitioning* assignment algorithm limits each core to a maximum of 9/16 of the total cache capacity and therefore the *lru_pointer* and the *Dirty Stack Distance* register sizes were set to 6 bits. The Hit and $Dirty$ counters size was set to 32 bits to avoid overflows during an epoch. Finally, we have implemented the LRU stack distance of the MSA as a single linked list with head and tail pointers. The cost for such a structure is included in Table 5.2. Overall, the implementation overhead is estimated to be 117 kbits per profiler, which is approximately 1.4% of the 8MB LLC cache design assuming 8 profilers.

Chapter 5

Bank-aware Cache Partitioning for Multicore Architectures

As Chip-Multiprocessor systems (CMP) have become the predominant topology for leading microprocessors, critical components of the system are now integrated on a single chip. This enables sharing of computation resources that was not previously possible. In addition, the virtualization of these computational resources exposes the system to a mix of diverse and competing workloads. Cache is a resource of primary concern as it can be dominant in controlling overall throughput. In order to prevent destructive interference between divergent workloads, the last level of cache must be partitioned. In the past, many solutions have been proposed but most of them are assuming either simplified cache hierarchies with no realistic restrictions or complex cache schemes that are difficult to integrate in a real design. To address this problem, this dissertation proposes a dynamic partitioning strategy based on realistic last level cache designs of CMP processors. To evaluate the proposed scheme a cycle accurate, full system simulator based on Simics and GEMS was used, simulating an 8-core DNUCA CMP system. Results on such an 8-core system show that the proposed scheme provides on average a 70% reduction in misses compared to non-partitioned shared caches, and a 25% misses reduction compared to static equally partitioned (private) caches. Finally, the scheme achieved comparable results with previous schemes that assumed simplified, unrestricted cache designs.

5.1 Introduction

Chip Multiprocessors (CMP) integration has brought abundant on-chip resources that can now be shared in finer granularity among the multiple cores. Such sharing though has introduced chip-level contention and the need of effective resource management policies is more important that ever.

To efficiently exploit these resources, systems require multiple program contexts and virtualization has become a key player in this arena. Many small and/or low utilization servers can now be easily consolidated on a single physical machine [2][65][80], allowing higher utilization of the available resources with significant energy reductions. Such consolidation presents both opportunities and pitfalls to computer architects to best manage these once isolated resources on large CMP designs.

In such virtualization environments, workloads tend to place dissimilar demands on shared resources and therefore, due to resource contention, are much more likely to destructively interfere in an unfair way. Consequently, shared resources' contention become the key performance bottleneck in CMPs [6][20][23][37]. Shared resources include, but are not limited to: main memory bandwidth, main memory capacity, cache capacity, cache bandwidth, memory subsystem interconnection bandwidth and system power.

Among these resources, several studies have identified the shared last-level cache (L2 in this study) of CMPs as a major source of performance loss and execution inconsistency [20][58][6][14][38][46][63]. As a solution, most of the proposed techniques control this contention by partitioning the L2 cache capacity and allocating specific portions of it to each core or execution thread. There are both static [20][38] and dynamic partitioning [58][63][25] schemes available that use workload profiling information to make a decision on cache capacity assignment for each core/thread. All of the above

techniques are usually based on high-level system characteristic monitoring since low-level activity based algorithms such as LRU replacement fail to provide a strong barrier among workloads competing for shared resources.

In addition to the cache partitioning need, as wire delays are gradually becoming the most important design factor in cache architectures, designers have successfully used banking techniques [19][78] to mitigate the effects of increasing wire delays for short distances. Banked architectures are now the typical design direction for caches in both industry and academia. Such solutions though are still not efficient enough since wire delays between banks themselves are still an important performance bottleneck. An alternative solution to the wire delay problem, mainly used in academia is the Non-Uniform Cache Architecture (NUCA) designs [37]. NUCA is based on assuming non-uniform access latencies to all cache banks of a large L2 cache. The NUCA model, which was originally proposed for a single core, was later extended to a multicore CMP version named CMP-NUCA by Beckmann *et al.* [3]. In parallel, industry has also responded to wire delay dominance of on-chip caches with non-uniform structures. These structures have been implemented with a small number of cache levels, rather than large arrays of homogeneous networks of cache blocks as are assumed in academia. Both approaches are logically similar and the differences are more tied to the physical implementation constraints of cache banks and data networks rather than higher-level policy options. As new CMP designs include more cores and cache capacity, a banked L2 cache design is a promising solution that can scale with the number of cores and is able to alleviate wire delay problems.

This dissertation highlights the problem of sharing the last level of cache in CMP systems and motivates the need for low overhead, workload feedback-based hardware/software mechanisms that can scale with the number of cores, for monitoring and controlling the L2 cache capacity partitioning. The need to address the dominating

effect of wire delays by taking into consideration the realistic constraints imposed by banking architectures drove the baseline system structure. Specifically, we propose a *Bank-aware* partitioning strategy for the CMP-DNUCA architecture, consistent with the current industry trends, that is aware of the banking structure of the L2 cache. To evaluate the partitioning scheme, we integrated an 8-core CMP system with a 16-way banked DNUCA L2 cache design, using Simics [48] combined with GEMS [49] full system, cycle accurate simulation toolset. In summary, the contributions of this chapter are the following:

1. A cache partitioning scheme is proposed, named *Bank-aware*, for CMP-DNUCA that is aware of the banking structure of the L2 cache, which show a 70% reduction in misses compared to non-partitioned shared caches, and a 25% misses reduction compared to static even partitioned (private) caches. Such miss rate reductions result in 43% and 11% reductions in CPI over the non-portioned and static even partitioned schemes, respectively.

2. A detailed implementation of a dynamic cache partitioning algorithm using a non-invasive, low-overhead monitoring scheme based on Mattson's stack distance algorithm . The overall hardware overhead for the proposed cache profiling scheme is equal to 0.4% of the baseline L2 cache design.

5.2 CMP-Baseline

Prior works in industry and academia have proposed quite varied allocation and migration schemes for the memory cache hierarchy. A large amount of work in academia has focused on free form, highly banked, and non-uniform cache structures. This was in response to the expected wire dominant nature of future technologies, where the latency of large monolithic caches would become detrimental to system performance. These

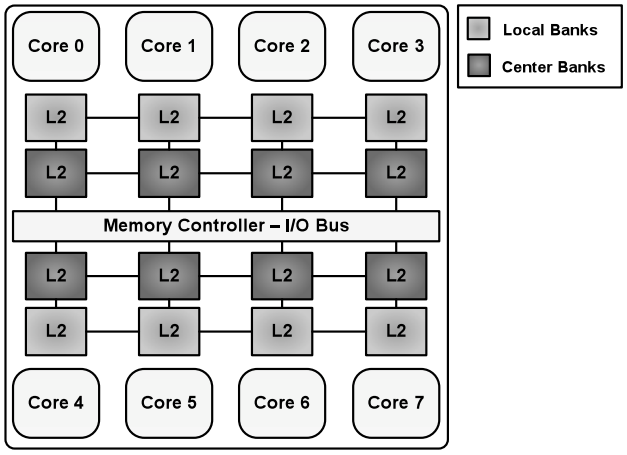


Figure 5.1: Baseline CMP architecture using a DNUCA last-level cache system that is assumed for the Bank-aware scheme.

proposed free form caches enable great freedom in allocation and migration policies. As an example Huh *et al.* in [25] proposed a 256 x 64K bank cache. In contrast, industry has thus far typically implemented more traditional structures with fewer than eight cache banks that form specific multi-level caches (compared to more free form NUCA like levels). For example, the recently announced 45nm Intel Nehalem processors has three levels on chip cache (32KB, 256KB, 4-8MB) [9], compared to two levels in the previous design. Such additional cache levels approach a more NUCA-like cache, formed of homogeneous cache banks.

The industry direction of avoiding highly banked structures can be also explained by recent upgrades to the CACTI 6.0 tool [54]. In this work, the authors demonstrated, using detailed modeling of the cache and interconnection subsystem, results with a remarkably different outcome from what academia assumes. As a case study, they evaluated a 32 MB L2 cache. This gave a mix of ideal cache block sizes of 4 MB and 8MB. This landscape drove the baseline system structure. Specifically, we limited the

Table 5.1: DNUCA-CMP parameters used for evaluation of the Bank-aware scheme.

Memory Subsystem		**Core Characteristics**	
L1 Data & Inst. Cache	64 KB, 2-way set associative, 3 cycles access time, 64 Bytes cache block size	Clock Frequency	4 GHz
L2 Cache	16 MB (16 x 1MB banks), 8 ways set associative, 10-70 cycles bank access, 64 Bytes cache block size	Pipeline	30 stages / 4-wide fetch / decode
Memory Latency	260 cycles	Reorder Buffer / Scheduler	128/64 Entries
Memory Bandwidth	64 GB/s	Branch Predictor	Direct YAGS / indirect 256 entries
Memory Size	4 GB of DRAM		
Outstanding Requests	16 requests / core		

total bank structures on the chip to 1MB per cache bank. This was chosen as the smallest reasonable bank size.

Fig. 6.1 shows the 8-core CMP-NUCA baseline system assumed in this study. The design uses as the last-level of cache a DNUCA L2 cache with 16 physical banks that provide a total of 16MB of cache capacity. Each cache bank is configured as an 8-way set associative cache. Another way to understand the configuration of the cache is as a 128-way equivalent cache that is separated in 16 cache banks of 8 ways each. The eight cache banks that are physically located next to a core are called *Local banks* and the rest are characterized as *Center banks*. Cores located next to *Local banks* have the minimum access latency but that delay can significantly increase when a core needs to access a *Local bank* physically located next to another core. *Center banks* have, on average, higher access

latency than *Local banks* but their distance from each core has smaller variation than *Local banks* and so does the access latency. The access latency to a L2 cache bank varies from 10 up to 70 cycles depending on the physical location of both the core requesting the access and the L2 bank containing the data. A core physical located next to a *Local* cache bank has to wait 10 cycles to access the bank. The maximum possible latency, assuming a moderate network contention, is equal to 70 cycles (i.e core 0 to access the *Local bank* next to core 7 that requires 7 hops). Overall, Table 8.2 includes the basic system parameters that have been selected for the baseline system.

5.3 Bank-aware Cache Partitioning

This section elaborates on the proposed *Bank-aware* cache partitioning scheme. First, this section provides details about the application profiling mechanism followed by the partitioning algorithm for assigning cache capacity to each core. In the end, the cache partitions allocation algorithm is described for allocating the cache partitions on the CMP-baseline system.

5.3.1 Cache Profiling of Applications

In order to dynamically profile the cache requirements of each core, the scheme implements a cache miss prediction model based on Mattson's stack distance algorithm (MSA). The profiling mechanism is described in details in Section 4.1. Notice that this scheme includes only the profiling mechanism to project the last-level cache misses as a function of the cache capacity allocated to each core and not the circuit to estimate the memory bandwidth.

The hardware overhead of the profiling structure is primarily defined by the implementation of the necessary cache directory tag shadow copy. These cache block

Table 5.2: Overhead of the proposed MSA profiler.

Structure Name	**Overhead Equation**	**Overhead**
Partial Tags	$tag\_width * ways * cache\_sets$	54 kbits
LRU Stack Distance Implem.	$((lru\_pointer\_size * ways) + head/tail) * cache\_sets$	27 kbits
Hit Counters	$cache\_ways * hit\_counter\_size$	2.25 kbits

tags are necessary for identifying which cache block is assigned at each one of the *hit* counters of Fig. 4.1 and allow a detailed monitoring of resource requirements on a cache block granularity on the last-level cache. Additional overhead is introduced by the implementation of the *hit* counters themselves for each cache way, but since those counters are shared over all the available cache-ways, their overhead is significantly lower than the cache block tag information for every set.

As explained the Section 4.4, to reduce the overhead one can use *a*) *partial tags* [35] *b*) *set sampling* [34] and *c*) *maximum assignable capacity* reduction techniques. In this dissertation the proposed implementation is based on all of the above methods. The overhead analysis showed that the use of 12 bit *partial tags* combined with 1-in-32 *set sampling* produced error rates within 5% of the profiling accuracy obtained using a full tag implementation. In addition, the proposed *Bank-aware partitioning* assignment algorithm limits each core to a maximum of 9/16 of the total cache capacity. The hardware overhead of the proposed implementation for every necessary structure is included in Table 5.2. Overall, the implementation overhead is estimated to be 83.25 kbits per cache profiler, which is approximately 0.4% of the 16MB last-level cache design for all the profilers.

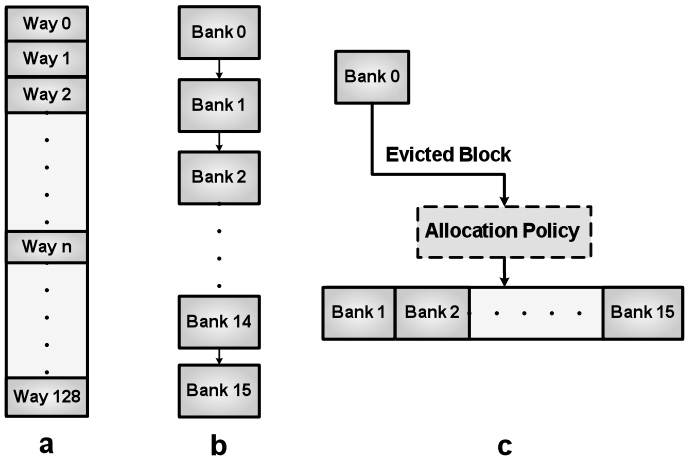


Figure 5.2: Last-level cache banks aggregation schemes.

5.3.2 Bank-aware Assignment of Cache Capacity

Prior work in MSA-based cache partitioning was analyzed on fully configurable caches shared among a small number of CPUs [23][63]. This type of partitioning algorithm will be named as *Unrestricted* for the remaining of the chapter. On the other hand, while Huh *et al.* in [25] proposed a method for partitioning a CMP-NUCA cache that relied on a highly banked structure that, as explained in Section 5.2, features an unrealistic physical implementation. As a solution, a method to partition cache bank structures is proposed using a MSA-based profiling mechanism aligned with current industry directions, that is, using a smaller number of higher capacity cache banks. Such configuration limits the granularity of possible partitions and imposes a set of restrictions over the *Unrestricted* techniques proposed in the past. This is rooted in the need to aggregate multiple cache banks into a single partition. The potential aggregation methods, shown in Fig. 5.2, are the following:

1. **Cascade:** In this approach, all cache banks that contain portions assigned to a given core are connected head to tail. To match the MSA Least Recently Used (LRU)

strategy (Fig. 5.2.a), all allocations are placed as Most Recently Used (MRU) at the head of the chain. Each allocation causes a shift down of the LRU. Evictions are passed down the chain from LRU out to MRU position in the next bank until a free spot is located (potentially formed from the cache hit that was moved to the top). This structure is shown in Fig. 5.2.b. This method provides for an LRU policy (assuming the banks are also LRU). The advantage of this method is that one can stitch together arbitrary fractions of banks which will emulate the MSA very closely. The primary disadvantage is the costly, high migration rate between cache banks.

2. **Address Hash:** A common approach to cache bank aggregation is the use of an *address hash*. Typically this method is used with a power of two number of cache banks, such that lower order address bits can directly select the bank. While systems have also been built with non-power of two hashes, it requires complex modulo operations in the hardware hash function. An example would be the IBM POWER4 and POWER5 processors, which hash across three banks [79]. In addition, Gao *et al.* [12] and Seznec *et al.* [68] proposed non-power of two hashing functions with increased complexity over simple hashing functions. Irrespective of the number of cache banks aggregated, this method requires symmetry in that each hashed bank must have the same cache capacity. As such, this method has some restrictions. Lastly, address hash features low migration rates.

3. **Parallel:** This method is very much like *Address Hash*, except that a line can be stored in any of the cache banks. Allocation is controlled by round robin/random selection. As such, any given line can be stored in any of the cache banks. This forces additional look-up operations in the directory structure (which is implemented as partial tags). This is less restrictive than *Address hash* in bank configuration, in that, non-power of 2 aggregations of banks are possible without

complex modulo address computations. The migration rate is equivalent to *Address Hash*, however, power is higher due to wider directory look-ups.

Even though *Cascade* provides the greatest flexibility, the migration rates observed in simulation are prohibitively high. Both *Address hash* and *Parallel* provide reasonable solutions to aggregating cache banks. The only restriction is that multiple banks must have the same capacity. The provide analysis will demonstrate that the degradation can be mitigated using the structure shown in Fig. 5.2.c. In this structure the level of cascading is limited to two. The allocation policy can be either *Address Hash* or *Parallel*. In the current analysis the *Parallel* approach is assumed.

These issues present problems in direct application of currently proposed *Unrestricted* cache partitioning schemes [63] and as a solution this chapter proposes a *Bank-aware* assignment algorithm. This algorithm is based on progressive control of bank granularity. Essentially, as the capacity assignment increases, small deviations from the ideal assignment are tolerable with respect to overall miss reduction. Based on this observation and the bank aggregation requirements, the following policies are proposed:

1. *Center* cache banks are completely assigned to a specific core. This prevents situations where aggregated banks are of different capacities.

2. Any core that is allocated *Center* banks, will receive a full *Local* bank.

3. *Local* cache banks can only be shared with an adjacent core. We only allow per assignment control at *Local* cache banks. In addition, requiring adjacent sharing provides for low latency and minimal network loads for data transfers.

A typical allocation is shown in Fig. 5.3. From the figure, most of the cores have multiple L2 cache banks allocated to them except core 2 and core 5. Those cores share the capacity of a single L2 bank with core 3 and 4, respectively.

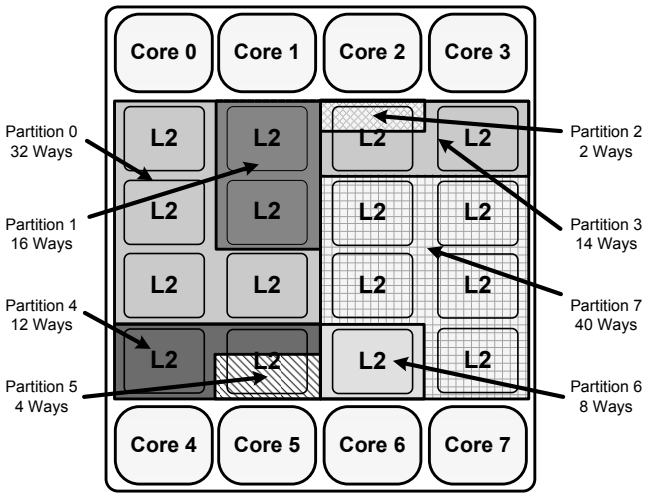


Figure 5.3: Example of a CMP cache capacity partitioning using the "Bank-aware" scheme.

To enforce the selected cache partitions, the typical design of a cache bank has to be modified to support a vertical, fine-grain, cache-way partitioning scheme, as was proposed in [23]. According to this scheme, each cache-way of a set associative cache can belong to one or more specific cores. When a specific core suffers a cache miss, a modified LRU policy is used to select the least recently used cache block among the ones that belong to that specific core, for replacement. Therefore, only cache-ways that belong to a specific core or set of cores can be accessed and the rest of the cache-ways that belong to other cores are not affected, eliminating the destructive interference between different workloads running on different cores. To reduce the design complexity, all of the sets in a cache bank are vertically partitioned with the same cache-ways assignment and therefore the granularity of assigning a different cache-way partition is a single cache bank.

5.3.3 Allocation Algorithm on CMP

This section describes in details the proposed *Bank-aware* assignment algorithm. The assignment policy is based on the concept of *Marginal Utility* [81]. This concept originates from economic theory, where a given amount of resources (in the current case cache capacity), provides a certain amount of utility (reduced misses). The amount of utility relative to the resource is defined as the *Marginal Utility*. Specifically, *Marginal Utility* is defined as:

$$Marginal\ Utility(n) = \frac{Miss\ Rate(c+n) - Miss\ Rate(c)}{n} \tag{5.1}$$

The MSA histogram provides a direct way to compute the *Marginal Utility* for a given workload across a range of possible cache allocations. We use this capability to make the *best* use of the limited cache resources. We follow an iterative approach, where at any point we can compare the *Marginal Utility* of all possible allocations of unused capacity. Of these possible allocations, the maximum *Marginal Utility* represents the *best* use of an increment in the assigned capacity.

The algorithm arrives at a capacity assignment via successive steps determining the maximum *Marginal Utility* for a subset of processors and assignment restrictions. The overall flow is shown in Fig. 5.4. The first step of the algorithm is to assign the cache-ways in *Center* cache banks. Following that, in Box 1, the maximum *Marginal Utility* is calculated and cache banks are assigned accordingly. For the calculations, we assume that each *Local* bank is assigned to the associated processor. In Box 2, we check if all the banks are assigned, if not, step 1 is repeated. Following Rules 1 and 2, we mark all processors with *Center* banks complete (Box 3). The next steps are used to solve the *Local* cache bank partitions. In Box 4, we once again find the maximum *Marginal Utility*, but assignments are limited to possible pairs of processors (in keeping with Rule 3). In Box 5, we check

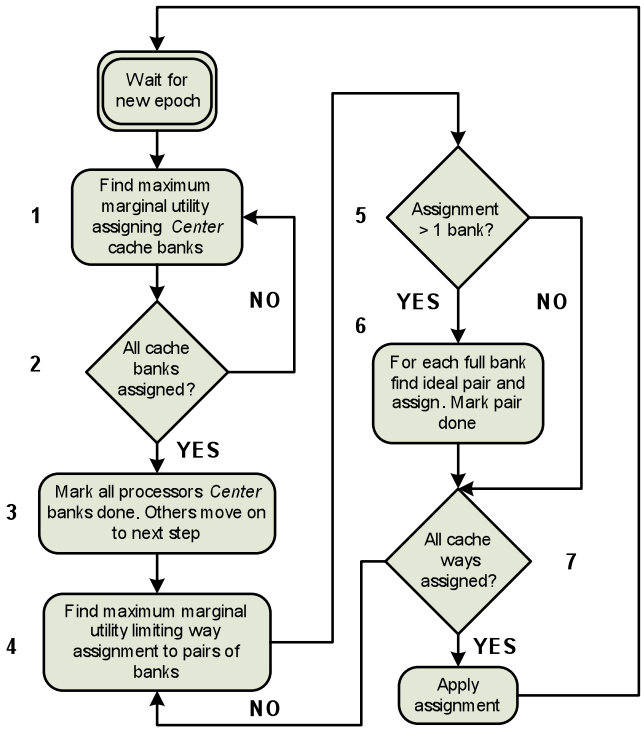


Figure 5.4: Flow chart of "Bank-aware" cache capacity allocation algorithm.

if the new assignment has caused any processor to overflow into another processors *Local* region. If so, we find the ideal pair with respect to minimal misses. Essentially we defer the pairing as many steps as possible, and make the best pairing choice once it is decided a processor should receive a fraction of an adjacent *Local* bank. Once the pair is assigned, both processors are marked complete. This step is repeated until all the cache ways are assigned.

5.4 Evaluation

To evaluate the proposed scheme we utilized a full system simulator, modeling an 8-core SPARCv9 CMP under Solaris 10 OS. Specifically, we used Simics [48] as the

full system functional simulator extended with the GEMS toolset [49] to simulate an out-of-order processor and memory subsystem. The CMP-NUCA design was implemented in GEMS' memory timing model (Ruby) extended with the support of the fine-grain L2 cache partitioning scheme described in Section 5.3. The memory system timing model includes a detailed message-based model of the inter-chip network using a MOESI cache coherence protocol. Throughout the proposed scheme in this chapter, the frequency of evaluating and reallocating the L2 cache partitions was set to a 100M cycle epoch.

This dissertation uses SPEC CPU2000 [59] scientific benchmark suite, compiled to SPARC ISA with peak configurations, as the workload of the proposed scheme. We fast forward all the benchmarks for 1 billion instructions, and use the next 100M instructions to warm up the CMP-NUCA L2 cache. Each benchmark was simulated in the CMP-NUCA using GEMS for a slice of 200M instructions after cache warm up. Table 8.2 includes the basic system parameters that were used.

5.4.1 *Bank-aware* vs. *Unrestricted* Partitioning

The analysis of computer systems in virtualization environments is an open problem. In these systems an arbitrary mix of work may share a server at any given time. The general problem of performance analysis using benchmarks is greatly compounded by the possible combinations of workloads. In order to limit the state space, we base the evaluation on the workloads of the SPEC CPU2000 integer and floating point benchmark suites. Even with this limitation the possible combinations of the 26 workloads on the eight core target machine is very large and equal to:

$$C(num\_workloads + num\_cores - 1, num\_cores) = C(26 + 8 - 1, 8) \qquad (5.2)$$

which is $\sim$14 million possible workload combinations. Based on this very large state space, we employ a comparative Monte Carlo approach to the evaluation of the assignment algorithm.

Total system miss rates are estimated from projecting miss rates based on MSA data collected from the detailed system simulations. To accurately cover the workload state space would require far too many simulations points than are possible assuming a full simulation of all the cases. Instead, we used the estimated method here, combined with detailed simulations of a more manageable number of workload mixes as a second form of validation. The comparison methodology is as follows:

1. Collect MSA histograms for a mix of workloads. In the current case these are the 26 components from SPEC CPU2000.
2. From these 26 components, we randomly select (with repetition) 8 workloads.
3. Execute both the *Unrestricted* and *Bank-aware* partition algorithms.
4. Compare the MSA predicted miss rates between the two partition algorithms and the case of fixed partitions of 2MB per core.

The above process was executed for 1000 random workload assignments. For each workload set, we compared the MSA miss rate based on a fixed even share per core (16 ways for each core), the *Unrestricted* algorithm, and the *Bank Aware* algorithm.

In Fig. 5.5 we show the miss rate relative to the even partitions for the *Unrestricted* and *Bank-aware* algorithms. A ratio of one represents no reduction in misses compared to static fixed partitioning, while zero indicates all misses are removed with the MSA-based partitioning scheme. We have sorted the 1000 results with respect to the miss rate reduction of the *Unrestricted* scheme. The even partitions and *Unrestricted* essentially form a performance envelope. Ideally, all of the *Bank-aware* assignments would fall on the *Unrestricted* line, which is in general true except some outliners that achieved smaller miss rate reductions than *Unrestricted*. Both figures give an indication of the range of

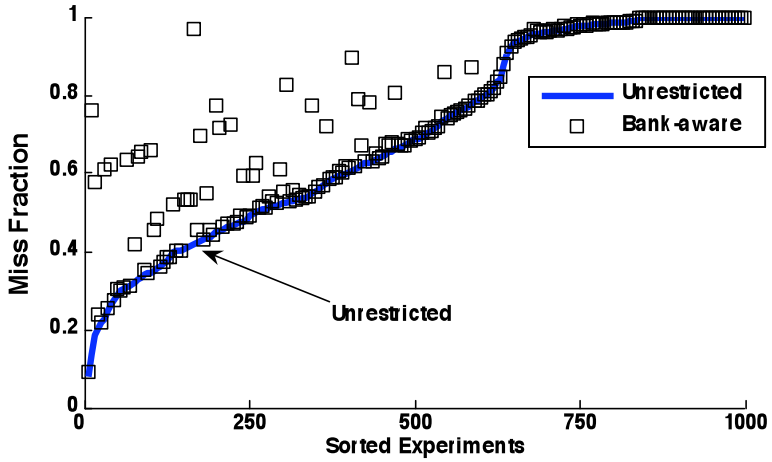


Figure 5.5: Relative miss rate compared to the fixed-share for *Unrestricted* scheme.

miss rate reductions possible. On average, the miss rate reduction from the *Unrestricted* and *Bank-aware* algorithms are quite comparable. The *Unrestricted* averages a 30% reduction in misses compared to 27% for the *Bank-aware* over the case of even partitions. This result shows that the restrictions placed on the allocation algorithm due to the more realistic implementation of L2 cache do not adversely affect the benefits of the MSA-based dynamic cache partitioning scheme.

5.4.2 Detailed Simulation Results

We randomly chose eight workload sets from the previous simulations to evaluate the proposed partitioning scheme on the 8-core full system shown in Fig. 6.1. Table 5.3 shows the selected workloads along with the cache-ways that were assigned to each core by the *Bank-aware* partitioning scheme. Fig. 5.6 and Fig. 5.7 shows the relative miss rate and CPI of *Equal-partitions* and *Bank-aware* partitioning over the simple case of *No-partitions*. *Equal-partitions* is equivalent to assigning private cache partitions of equal size to each core. From the figures, both partitioning schemes show a significant

Table 5.3: 8-core full system workload experiments.

Set	Benchmarks & "cache-ways" assignments from core0 to core7 [benchmark(#ways)]
1	apsi(12), galgel(4), gcc(2), mgrid(16), applu(16), mesa(8), facerec(56), gzip(8)
2	crafty(12), gap(4), mcf(24), art(16), equake(8), equake(8), bzip2(48), equake(8)
3	applu(12), galgel(4), art(16), art(16), sixtrack(16), gcc(6), mgrid(40), lucas(16)
4	mgrid(40), mcf(24), art(16), equake(16), gcc(6), equake(10), sixtrack(6), crafty(10)
5	facerec(56), fma3d(8), sixtrack(16), apsi(16), fma3d(6), ammp(10), lucas(6), swim(10)
6	bzip2(48), gcc(8), twolf(16), mesa(24), wupwise(6), applu(10), fma3d(6), ammp(10)
7	swim(8), parser(16), mgrid(40), twolf(16), fma3d(2), parser(14), swim(8), mcf(24)
8	ammp(13), eon(3), swim(11), gap(5), gcc(8), art(16), twolf(56), art(16)

reduction in misses and CPI over the simple *No-partitions* one, which is a strong indication of the need for partitioning the last level of cache. On average, *Bank-aware* shows a 70% and 43% reduction in misses and CPI over *No-partitions*, respectively. Moreover, from Fig. 5.6, the *Bank-aware* partitioning scheme shows on average a 25% reduction over simple *Equal-partitions*. This reduction is inline with the reduction estimated in the Monte Carlo experiment of the previous section. In addition, Fig. 5.7 shows that *Bank-aware* partitioning can achieve an 11% reduction in CPI over the *Equal-partitions* scheme. Comparing Fig. 5.6 and 5.7, some sets of workloads demonstrate a much higher performance sensitivity to misses than others since a reduction on L2 misses does not always result in an equal size reduction in CPI. For example, in *Set 1* even though we significantly reduced the overall fraction of misses, that reduction in not translated in CPI gain due to the overall small number of misses in that set and the performance characteristics of the applications that feature the highest miss reduction.

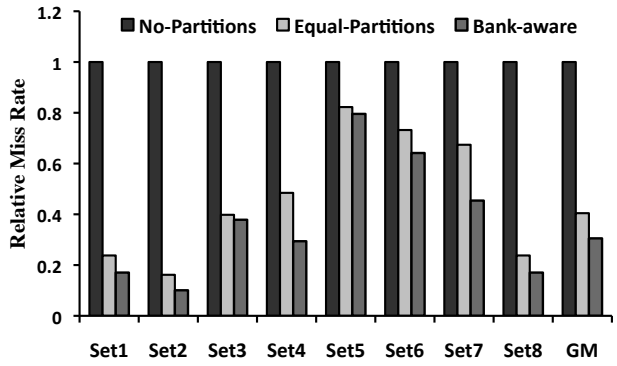


Figure 5.6: Relative miss rate of 8-core sets compared to the no-partitioning scheme.

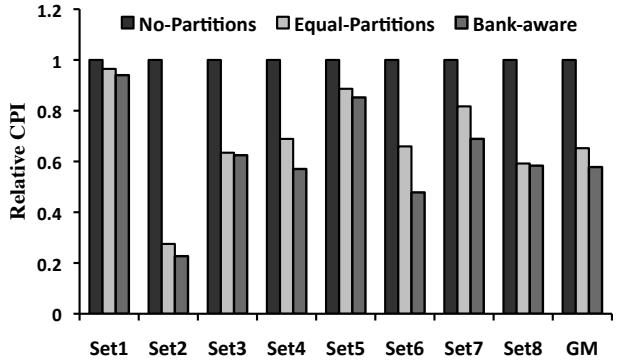


Figure 5.7: Relative CPI of 8-core sets compared to the no-partitioning scheme.

5.5 Summary

Shared resource contention in CMP platforms has been identified as a key performance bottleneck that is expected to become worse as the number of cores on a chip continues to scale to higher numbers. Many solutions have been proposed, but most assume either simplified cache hierarchies with no realistic restrictions or complex cache schemes that are difficult to integrate in a real design. Therefore, both approaches could lead to conclusions that are unrealistic when implemented in a real system. In this dissertation the problem of sharing the last level of cache in CMP platforms highlighted

and we motivate the need for a low-overhead cache partitioning scheme that is aware of the banking structure of the L2 cache design. The proposed *Bank-aware* partitioning scheme demonstrates a 70% reduction in misses compared to non-partitioned shared caches, and a 25% miss rate reduction compared to even partitioned (private) caches. Lastly, the proposed scheme managed, on average, the same miss reduction achieved with less realistic proposed *Unrestricted* schemes that are unaware of implementation restrictions.

Chapter 6

Bandwidth-aware Memory-subsystem Resource Management for Large CMP Systems

By integrating multiple cores in a single chip, Chip Multiprocessors (CMP) provide an attractive approach to improve both system throughput and efficiency. This integration allows the sharing of on-chip resources which may lead to destructive interference between the executing workloads. Memory-subsystem is an important shared resource that contributes significantly to the overall throughput and power consumption. In order to prevent destructive interference, the cache capacity and memory bandwidth requirements of the last-level cache have to be controlled. While previously proposed schemes focus on resource sharing within a chip, we explore additional possibilities both inside and outside a single chip. This dissertation proposes a dynamic memory-subsystem resource management scheme that considers both cache capacity and memory bandwidth contention in large multi-chip CMP systems. The presented approach uses low overhead, non-invasive resource profilers that are based on Mattson's stack distance algorithm to project each core's resource requirements and guide the cache partitioning algorithms. The described bandwidth-aware algorithm seeks for throughput optimizations among multiple chips by migrating workloads from the most resource-overcommitted chips to the ones with more available resources. Use of bandwidth as a criterion results in an overall 18% reduction in memory bandwidth along with a 7.9% reduction in miss rate, compared to existing resource management schemes. Using a cycle-accurate full system simulator, the scheme achieved an average improvement of 8.5% on throughput.

6.1 Introduction

Large, high performance CMPs systems are typically deployed in large server systems. These large systems utilize virtualization where many independent small and/or low utilization servers are consolidated [65]. Under such environment the resource sharing problem is extended from the chip-level to the system-level, and therefore system-level resources, like main memory capacity and bandwidth, have to be considered for the appropriate sharing policies. Consequently, to design the most effective future systems for such computing resources, effective resource management policies are critical not only in mitigating chip-level contention, but also in improving system-wide performance and fairness.

A great deal of research has recently been proposed on harnessing the shared resources at a single CMP chip level. The primary focus has been to control contention on the resources that most affect performance and execution consistency, that is the shared *last-level cache* capacity [14, 20, 25, 38, 46, 58, 63], and the *main memory bandwidth* [57]. To address last-level cache contention, the proposed techniques partition the last-level cache capacity and allocate a specific portion to each core or execution thread. Such private partitions provide interference isolation and therefore can guarantee a controllable level of fairness and QoS. There are both static [20, 38] and dynamic partitioning algorithms proposed [25, 46, 58, 63] that use profiled workload information to make a decision on cache capacity assignment for each core/thread. On the other hand, to address the memory bandwidth contention problem, *fair queuing* network principles [57] have been proposed. Those techniques are based on assigning higher priorities on bandwidth use either to the most important applications (for QoS) or to the tasks that are close to missing a deadline (for fair use). In addition, machine learning techniques have also been employed in managing multiple interacting resources in a coordinated fashion [4]. Such methods require a hardware implemented artificial neural network to dynamically configure the

appropriate partitions for different resources.

Prior work monitoring schemes can be classified into *trial-and-error* and *prediction-based* configuration exploration. *Trial-and-error* systems are based on observing the behavior of the system under various configurations while *predictive* systems are able to concurrently infer how the system will perform under many configurations. As the state space of the system grows, *predictive* schemes have inherent scalability advantages. Examples of representative *predictive* schemes that have been proposed in the past are the one from Zhou *et al.* [89] for main memory and Qureshi *et al.* [63] for last-level caches. Both are based on Mattson's stack distance algorithm [50] and are able to provide miss rate predictions of multiple last-level cache configurations in parallel. While these proposals provide reasonable speed-ups by solving the memory capacity assignment, they neglect the role of memory bandwidth constraints and system-level optimizations opportunities.

Virtualization systems are most effective when multiple CMP processors are aggregated into a large compute resource. In such environments, optimization opportunities exist in the ability to dispatch/migrate jobs targeting full system utilization. Limiting optimizations to a single chip can produce sub-optimal solutions. Therefore, while previous methods have shown good gains within a single chip, this dissertation explores additional possibilities in the context of large multi-chip CMP systems.

Overall, this chapter describes a dynamic, bandwidth-aware, memory-subsystem resource management scheme that provides improvements beyond previous single chip solutions. The solution aims at a global resource management scheme that takes into account: *a*) the limited available main memory bandwidth per chip and the performance degradation of the overall system when executing applications with over-commited memory bandwidth requirements, and *b*) the hierarchical nature of large CMP computer systems. In particular, the contributions of the presented in this chapter scheme are as follows:

1. A low overhead, non-invasive, hardware profiler implementation based on Mattson's stack distance algorithm (MSA) [50] is proposed that can effectively project memory bandwidth requirements of each core in addition to cache capacity requirements that previous papers have proposed [63]. The described implementation requires approximately 1.4% of the size of an 8MB last-level cache and introduces an *Effective error* of only 3.6% in bandwidth and 1.3% in cache capacity profiling. This MSA profiler can guide the proposed fine-grained dynamic resource management scheme and allow making projections of capacity and bandwidth requirements under different last-level cache partition allocations.

2. The scheme proposes a system-wide optimization of resource allocation and job scheduling. It aims to achieve overall system throughput optimization by identifying over-utilized chips, in terms of memory bandwidth and/or cache capacity requirements. For those chips, a set of job migrations is able to balance the utilization of resources across the whole platform leading to improved throughput. Based on the contacted evaluation, the bandwidth-aware scheme is able to achieve a reduction of 18% reduction in memory bandwidth along with a 7.9% reduction in miss rate and an average 8.5% increase in IPC, compared to single chip optimization policies.

6.2 Baseline System Architecture

Fig. 6.1 demonstrates an example of server construction in a large CMP environment where machine racks are filled with high density machines. In such systems, the servers are typically comprised of 2 to 4 packaged chips, with each chip being a CMP. In the baseline system, each CMP features 8 cores sharing a DNUCA-like [3] cache design as the last-level cache. Work can be partitioned and migrated as needed, based on scheduling and load balancing algorithms.

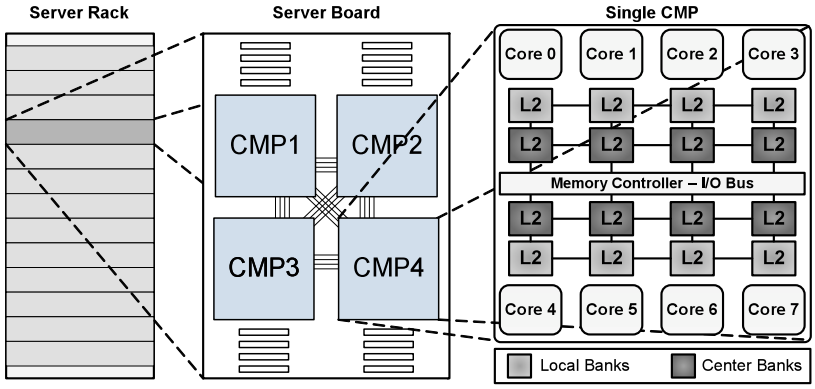


Figure 6.1: Baseline multi-chip CMP system assumed for the study of the "Bandwidth-aware" scheme.

The *"Single CMP"* part of Fig. 6.1 (*Single CMP*) illustrates the baseline CMP system that is assumed in this study. The selected last-level cache contains 16 physical banks with each cache bank configured as an 8-way set associative cache. The overall last-level cache capacity was set to 8MB. An alternative logical interpretation of the cache architecture is as a 128-way equivalent cache separated in sixteen 8-way set associative cache banks. The eight cache banks physically located next to a core are called *Local banks* and the rest are the *Center banks*. Table 6.1 includes the basic system parameters selected for the baseline system.

6.3 Bandwidth-aware Resource Management

This section elaborates on the proposed *Bandwidth-aware* cache partitioning scheme. First, there is a small description of the profiling mechanisms used to profile the applications' resource requirements. In the following, the section describes the proposed scheme that is divided in two basic algorithms: the *Intra-Chip* and the *Inter-Chip* resource partitioning algorithms. The overall scheme and how the two algorithms are combined

Table 6.1: Single-Chip CMP parameters used to evaluate the "Bandwidth-aware" scheme.

Memory Subsystem		**Core Characteristics**	
L1 Data & Inst. Cache	64 KB, 2-way associative, 3 cycles access time, 64 Bytes block size, Pseudo-LRU	Clock Frequency	4 GHz
L2 Cache	8 MB (16 x 512KB banks), 8-way associative, 10-70 cycles bank access, 64 Bytes block size, Pseudo-LRU	Pipeline	30 stages / 4-wide fetch / decode
Memory Latency	260 cycles	Reorder Buffer / Scheduler	128/64 Entries
Memory Bandwidth	64 GB/s	Branch Predictor	Direct YAGS / indirect 256 entries
Memory Size	4 GB of DRAM		
Outstanding Requests	16 requests / core		

together is described in the end of the section.

6.3.1 Applications' Resource Profiling

In order to dynamically profile the memory-subsystem resource requirements of each core, we implemented a prediction scheme that is able to estimate both cache misses and memory bandwidth requirements. The profiling mechanism for cache misses and memory bandwidth is described in details in Section 4.1 and Section 4.2, respectively.

As analyzed in Section 4.4, the overhead of the profiling mechanisms is estimated to be 117 kbits per profiler, which is approximately 1.4% of the 8MB last-level cache cache design assuming 8 profilers. The implemented profiling scheme uses 11 bit *partial hashed tags* (Tag hashing 2048) along with 1-in-32 *set sampling* and produces an average *Absolute*

error rates of 6.4% for misses and 5% for bandwidth. On the other hand, the *Effective error* was estimated to be 1.3% for misses and 3.6% for bandwidth compared to the profiling accuracy obtained using a full tag implementation. Such error bounds are inline with other set-sampling based monitor schemes like UMON [63] and CacheScouts [88]. The first one concludes that 1-in-32 *set-sampling* is enough for their profiler and the latter reports error rates of 6% for their 1-in-128 set sampling of cache occupancy for scientific workloads. To further improve overhead, the *Intra-chip partitioning* assignment algorithm limits each core to a maximum of 9/16 of the total cache capacity and therefore the *lru_pointer* and the *Dirty Stack Distance* register sizes were set to 6 bits.

6.3.2 Intra-Chip Partitioning Algorithm

As for the case of Chapter 5 (Section 5.3.3), the *Intra-chip* cache capacity assignment policy is based on the concept of *Marginal-Utility* [81]. The MSA histogram provides a direct way to compute the *Marginal-Utility* for a given workload across a range of possible cache allocations. The algorithm follows an iterative approach, where at any point it can compare the *Marginal-Utility* of all possible allocations of unused capacity. Of these possible allocations, the maximum *Marginal-Utility* represents the *best* use of an increment in assigned capacity. The greedy algorithm is shown in Algorithm 1. The general form of the algorithm appears in [73] and later on modified by Qureshi *et al.* in [63] for cache partitioning allocation.

Using Algorithm 1 as a guideline we have implemented a detailed **Intra-chip partitioning algorithm**. The algorithm takes into consideration the ideal capacity assigned by Algorithm 1 and the distributed nature of the DNUCA last-level cache design, to assign specific cache-ways to each core. The algorithm is based on a set of heuristics that are applied on the initial ideal assignments to decide on the partitions placement. The heuristics are the following [33]:

Algorithm 1 Marginal-utility allocation algorithm.

```
/* Initial */
num_ways_free = 128
best_reduction = 0
/* Repeat until all ways have been allocated to a core */
while (num_ways_free) {
   for core = 0 to num_of_cores {
        /* Repeat for the remaining un-allocated ways */
        for assign_num = 1 to num_ways_free {
          local_reduction = (MSA_hits(bank_assigned[core]
                  +assign_num) − MSA_hits(bank_assigned[core]))
                  /assign_num;
          /* keep the best reduction so far */
          if (local_reduction > best_reduction) {
                best_reduction = local_reduction;
                save(core, assign_num);
          }
        }
   }
   retrieve(best_core,best_assign_num);
   num_ways_free − = best_assign_num;
   bank_assigned[best_core]+ = best_assign_num;
}
```

1. *Center* cache banks are completely assigned to a specific core. This enables efficient aggregations of multiple cache banks.
2. If a core is assigned less than 8 ways (the size of a cache bank), all the ways are assigned to a *Local* bank close to it to keep a low access latency for the core.
3. *Local* cache banks can only be shared with an adjacent core in order to provide low latency and minimal network loads for data transfers.

To enforce the selected cache partitions, we modified the typical design of a cache bank to support a fine-grain, cache-way partitioning scheme as was proposed in [23]. According to this scheme, each way of a cache bank can only belong to one specific core.

When a core suffers a cache miss, a modified LRU replacement mechanism is used to select the least recently used cache block that belongs to that specific core, for replacement. This approach is compared against the scheme from Qureshi *et al.* in [63] enhanced for the CMP DNUCA last-level cache, which from now on we call UCP+ (Utility-based Cache Partitioning).

Each iteration of Algorithm 1 requires up to the number of available cache ways computations. As the cache ways are allocated, the number of ways a core can receive in each iteration reduces accordingly. Assuming the worst-case assignment of ways to be one cache way per iteration of the algorithm, its computational complexity is equal to $N + (N-1) + ... + 1 = N * (N-1)/2 = O(N^2/2)$, where N is the maximum allowable number of cache ways we can assign to a core. Moreover, since we assign cache ways in a granularity of 8-ways for most of the cases, for an equivalent of 128 ways last-level cache the $N \approx 128ways/8 \approx 16$. Therefore the complexity of estimating the cache partitions in this case is significantly less than the one needed by the Utility-Based algorithm of Qureshi *et al.* [63] for a large CMP system like the baseline. Finally, such complexity is less significant because of the low frequency of these computations. As the evaluation section shows, we use epochs of 100M cycles to re-evaluate the last-level cache partitions and as the evaluation is not part of the critical computation path it can be done one epoch later, minimizing the performance impact.

6.3.3 Inter-Chip Partitioning Algorithm

The **Inter-chip partitioning algorithm** utilizes the non-uniform marginal utilities of *Intra-chip partitioning algorithm* and the bandwidth requirements of each core to find an efficient workload schedule on the various available chips within the system. Given a random workload assignment among many CMP chips in a system, some chips are expected to feature higher contention of cache capacity and memory bandwidth than

others. To mitigate this problem, we propose a global partitioning algorithm that aims at lowering this system level contention. This approach first uses a global *Marginal-Utility* assignment to guide migrations of work between the chips in the system in an effort to relieve cache capacity contention. This optimization step is combined with memory bandwidth over-commit detection, which can potentially create additional migrations. Each migration step is evaluated against a heuristic so that the migration overhead is bounded with respect to the expected execution speed gains. In the following we describe the two basic steps of the algorithm.

6.3.3.1 Cache Capacity

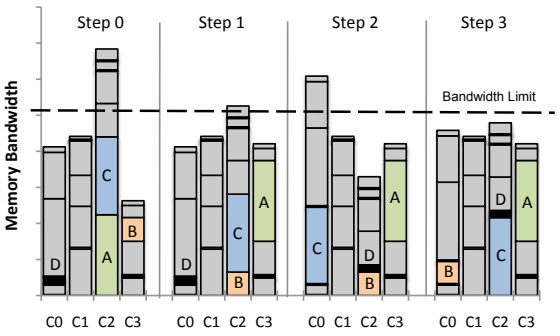
First, we use the *Marginal-Utility* allocation of Algorithm 1 to estimate an optimal resource assignment for each core (*ideal*), assuming that each core can freely use all of the available cache capacity in any chip. This gives us an optimal resource assignment per core and allows us to estimate the distance of the *Intra-chip partitioning algorithm* assignment from the ideal one per core. Having this information, we use Algorithm 2 to perform workload swaps between chips following a greedy approach. In line 1 of the algorithm, we estimate the distance, in number of cache ways, of each core's capacity assignment from its *ideal* one. Based on that we find the core with the worst cache-ways assignment in each chip. Lines 3 and 4 find the chip and core that has the biggest number of surplus ways in the system, respectively. The surplus ways are the ways assigned to a core that do not significantly contribute to the miss rate of the core's workload, based on the MSA profiler and their *Marginal-Utility* analysis, and therefore can be reassigned with small performance cost. This greedy approach swaps (migrates) the workloads of the worst and best identified cores (line 5) and re-estimates the overall miss rate of the current assignment. The whole process is repeated until the threshold based on migration cost is reached (line 7).

Algorithm 2 Inter-Chip Capacity-based Workload Swapping.

```
    do {
      for-each core in system-cores {
1           estimate core partition efficiency over ideal
2           find worst-core ∈ system-core with largest deficiency
      }
      for-each chip in system-chips {
3           find best-chip ∈ system-chips with largest surplus of ways
4           find best-core in best-chip with lowest capacity requirement
      }
5     swap workloads of best-core with worst-core
6     estimate overall_miss_rate
7   } while (overall_miss_rate − previous_miss_rate < threshold)
```

6.3.3.2 Memory Bandwidth

Using the proposed MSA-based profiler we can predict the bandwidth requirements of a given cache capacity assignment of each workload. An efficient assignment of cache partitions by the *Intra-chip partitioning* algorithm may end up having a very high memory bandwidth requirement per chip. Such an assignment can therefore saturate the available bandwidth. Note that the latency due to bandwidth over-commit is non linear, typically following an exponential relation with the network utilization. The migration based **Memory Bandwidth Over-commit** algorithm attempts to find combinations of workloads with high/low bandwidth requirements. The workload with higher bandwidth demands are shifted from over-committed chips to under-committed chips. The swapping workloads must have similar cache capacity assignments, within a small percentage difference (10% in case under study), in order to allow such swapping. This is necessary to guarantee that the migrations caused by the *memory bandwidth over-commit* algorithm do not contradict the assignments of the *Intra-chip cache partitioning* algorithm. The process is repeated up to the point there is either no bandwidth over-commited chips or additional swapping does not offer any reduction in bandwidth usage. Fig. 6.2(a) shows an



(a) Memory Bandwidth Over-commit algorithm illustration.

Step #	**Workload**($Chip_{Initial} \rightarrow Chip_{Final}$)
Step 0	Initial step, no swaps
Step 1	A($C2 \rightarrow C3$), B($C3 \rightarrow C2$)
Step 2	C($C2 \rightarrow C0$), D($C0 \rightarrow C2$)
Step 3	C($C0 \rightarrow C2$), B($C2 \rightarrow C0$)

(b) Workload swaps per step.

Figure 6.2: Four steps example of *Memory Bandwidth Over-commit* algorithm for four chips (C0 to C3).

example of how *Memory Bandwidth Over-commit* algorithm works on a four-chips system case with each chip having 8 randomly selected workloads executing on it from SPEC CPU2006 suite [16]. The Y-axis shows the stacked memory bandwidth requirements per chip as it was projected by the MSA-based profiler. The figure includes four steps of the algorithm (steps *Step 0* to *Step 3*). In the initial step *Step 0*, *Chip 2* (C2) is assigned a set of workloads which memory bandwidth requirements exceed the available one. If this set of workloads is executed on a single chip, the memory bandwidth contention will slow down the chip, affecting the overall throughput. The selected workloads are: *lbm* (A), *calculix* (B), *bwaves* (C) and *zeusmp* (D). Note that *zeusmp* (D) has very small memory bandwidth requirements. Table 6.2(b) shows the migrations that the algorithm performs in order to find a solution that satisfies the maximum bandwidth constraint. As we can see from *Step 3*, the algorithm terminates with an assignment that meets the bandwidth restriction for every chip.

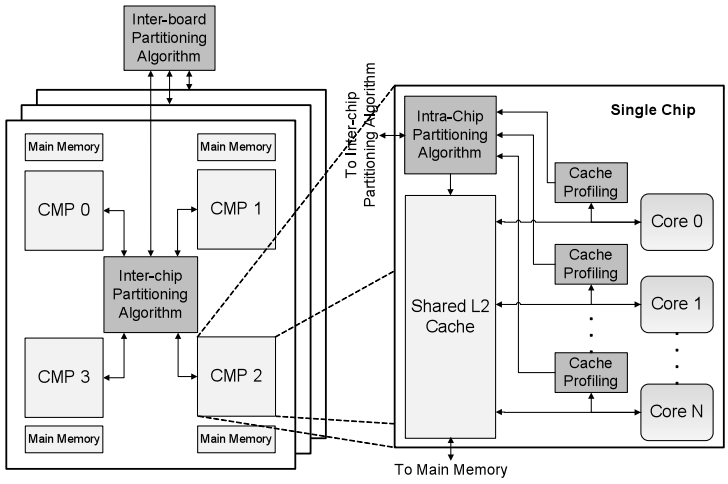


Figure 6.3: Presented"Bandwidth-aware" resource management framework.

6.3.3.3 Computational Overhead of Inter-chip algorithm

The *Inter-chip* algorithm is based on both Algorithm 1 and Algorithm 2 with the first being more computational demanding. Therefore, the computational complexity is bounded by the one of the *Marginal-Utility* algorithm which according to Section 6.3.2 is equal to $O(N^2/2)$. In addition, Algorithm 2 investigates migration only among the chips that are over-utilizing the memory bandwidth and the number of steps is limited by a threshold. As in the case of the *Intra-chip* algorithm, such computational complexity is less significant because of the low frequency of computation. The evaluation take place infrequently and can be done off-line, minimizing the performance impact.

6.3.4 Overall Dynamic Scheme

Fig. 6.3 shows an outline of the framework for the proposed hierarchical bandwidth-aware resource management scheme. The dark shaded modules indicate the additions over a typical large CMP system as the one described in Section 6.2. Looking at the single-chip

level, each core has a dedicated *Cache Profiling* circuit that tracks its shared resource (cache capacity and memory bandwidth) requirements (description in Section 7.3.1). This profiling circuit is independent of the cache subsystem and is able to non-invasively monitor the behavior of an application running on a core. Each *Cache Profiling* circuit is unaware of the workloads executing on other cores and assumes that the whole cache is available to the monitoring core. The overall proposed dynamic scheme is based on the notion of epochs. During an epoch, the MSA-based profiling circuit constantly monitors the behavior of each core in every CMP of the overall system. When an epoch ends, the profiled data of each core are passed to the *marginal-utility* algorithm to find the ideal cache capacity assignments for each core in every individual CMP-chip. Those partitions are then given to the *Intra-chip cache partitioning* algorithm for assigning specific cache-ways to each core using the heuristics described before. When each core has a specific cache partition assigned to it, we look at a higher than a single chip level for further optimizations in both cache capacity and memory bandwidth usage. To accomplish that we use the *cache capacity* algorithm of the *Inter-chip partitioning scheme* to find better cache allocations among multiple chips. Following that, we use the *Memory bandwidth* part of the same scheme to identify and solve bandwidth over-committed chip. In the end, we perform the necessary workload migrations and update the assigned cache partitions to each chip. The whole process is repeated at the end of each epoch.

6.4 Evaluation

To evaluate the proposed scheme we utilized a full system simulator, modeling an 8-core SPARCv9 CMP under Solaris 10 OS. Specifically, we used Simics [48] as the full system functional simulator extended with GEMS toolset [49] to simulate a cycle-accurate out-of-order processor and memory subsystem. The CMP-NUCA design was implemented in GEMS memory timing model (Ruby) extended with the support of the

fine-grain L2 cache partitioning scheme described in Section 6.3. The memory system timing model includes a detailed, inter-core network using a MOESI cache coherence protocol. Throughout this work, the frequency of evaluating and reallocating the L2 cache partitions on a simple chip was set to a 100M cycle epoch.

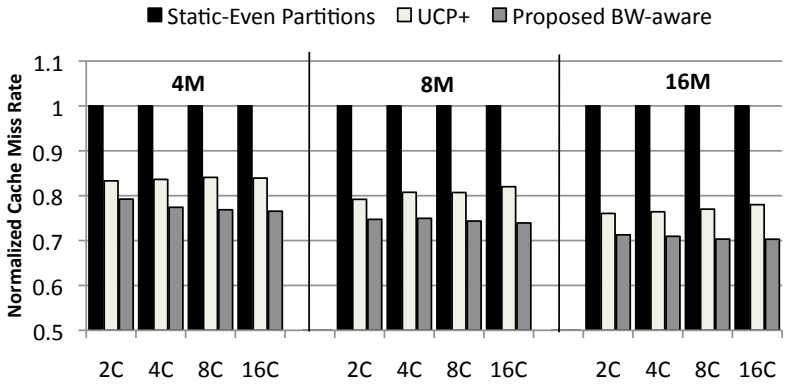
We use SPEC CPU2006 [16], SPLASH-2 [84], SPEC jbb2005 [60] and Apache2 [11] scientific and commercial benchmark suites, compiled to SPARC ISA with peak compilation options, to evaluate the proposed scheme. We fast forward all the benchmarks for 1 billion instructions, and use the next 100M instructions to warm up the CMP-NUCA L2 cache. Each benchmark was simulated in the baseline CMP-NUCA for a slice of 200M instructions after cache warm up. For SPLASH-2 we used the special "magic" instructions provided by our simulation tools to indicate the beginning and ending of the computational kernel of each workload. In that case, we warm-up the caches up to the first magic breakpoint and simulate with our detailed simulator until the all of the second ending breaking point are reached. Finally, for the case of SPECjbb2005 we warmed up the caches for 100K transactions ramping up to an appropriate number for warehouses (parallel threads) and simulate for the following 1k transactions. Finally for Apache2 we scale the server to the appropriate number of threads and warm up cache for the first 10K HTTP requests per Apache thread used. For our evaluation we simulated in our detailed simulator 100 requests per thread. For more information regarding our Apache2 setup please read 3.3.4 Table 6.1 includes the basic system parameters that were used. In the remaining of the evaluation we provide evaluation and comparison of the proposed scheme over one of most used cache partitioning algorithms in the recent literature, the *Utility-based Cache Partitioning* [63], extended to match the baseline system (*UCP+*).

Analysis of computer systems in virtualization environments is an important problem. In these systems an arbitrary mix of work may share a server at any given time. The general problem of performance analysis using benchmarks is greatly compounded

by the possible combinations of workloads. In order to limit the state space, we base the evaluation on the workloads of the SPEC CPU2006 integer and floating point benchmark suites. Even with this limitation the possible combinations of the 29 workloads on an eight core target machine is very large and equal to $C(num\_workloads + num\_cores$ - $1, num_cores) = C(29 + 8\text{-}1, 8) \approx$30 million possible workload combinations. Based on this very large state space, we employ a Monte Carlo based approach for the evaluation of the proposed scheme. The cache miss rates for the overall system are estimated from projecting miss rates based on MSA data collected from the baseline detailed system simulations. We used this method, to evaluate the proposed scheme over a range of cache sizes and number of chips in the system. The evaluation method was executed for 1000 random workload assignments for each configuration and includes the following steps: a) Collect MSA histograms from detailed simulations for all workloads, b) Randomly select (with repetition) a workload for each core in the simulated system and c) Execute stand-alone *Intra-chip* algorithm (*UCP+*) and the proposed *Bandwidth-aware* algorithm (*BW-aware*), that is *UCP+* combined with *Inter-chip* algorithms together. In the following we first report the gains in last-level cache misses followed by the gains in bandwidth use.

6.4.1 Last-level cache misses

Fig. 6.4 shows the normalized cache miss rate results from 2 to 16 CMP chips and last-level cache sizes of 4, 8 and 16MB. The first bar of Fig. 6.4(a) (*Static-Even Partitions*) represents the miss rate for the static even partitions where each core is statically allocated 1/8 of the cache capacity. The next two bars show the relative miss rate of the *UCP+* and *BW-aware* partitioning schemes, respectively. As we stated before, the *UCP+* is my implementation of the *Utility-based Cache Partitioning* algorithm proposed by Qureshi *et al.*[63] extended with a set of heuristics to support an 8-core CMP-DNUCA system like the baseline one. According to the figure, the average reduction provided by *BW-*



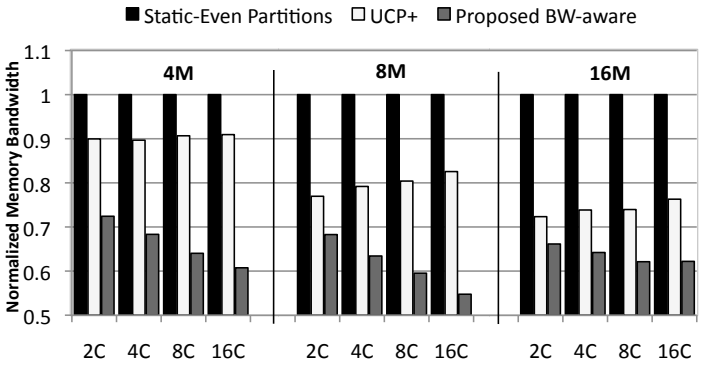
(a) Relative miss rate for different number of chips (2 to 16 chips) and sizes of last-level cache (4 to 16MB).

	Cache Size per Chip		
# of Chips	**4MB**	**8MB**	**16MB**
2 Chips	4.9%	5.6%	6.2%
4 Chips	7.1%	7.4%	7.5%
8 Chips	8.5%	7.8%	8.6%
16 Chips	8.7%	9.8%	9.9%

(b) Average relative miss rate reduction of presented Bandwidth-aware scheme over UCP+.

Figure 6.4: Relative miss rate improvement of UCP+ and presented Bandwidth-aware over Static-even partitioning scheme.

aware is 25.7% over the simple static-even partitions scheme. This is a significant reduction considering the relatively small 1.4% storage overhead of the MSA-based profiling structures. In addition, Fig. 6.4(b) shows the additional miss rate reductions provided by *BW-aware* algorithm over the *UCP+*'s scheme. From the figure, we can see an average last-level cache misses reduction of 7.9% taking into consideration all the configurations. Therefore, the *BW-aware* scheme provides significant additional miss rate reductions with no significant additional hardware over *UCP+*. As the number of chips in the system increases, the *BW-aware* algorithm shows improvement in the miss rate up to 8 chips after which we observe diminishing returns. This indicates that large SMP systems



(a) Relative memory bandwidth for different chip configurations.

	Cache Size per Chip		
# of Chips	**4MB**	**8MB**	**16MB**
2 Chips	17.5%	8.6%	6.1%
4 Chips	21.3%	15.5%	19.6%
8 Chips	26.6%	20.8%	11.8%
16 Chips	30.1%	27.8%	14.1%

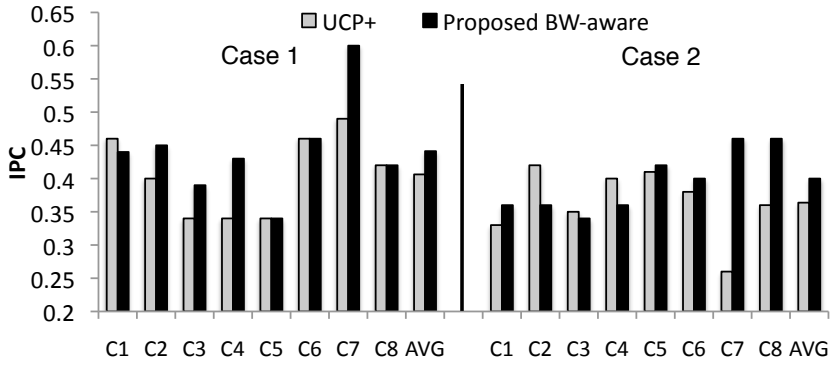
(b) Average bandwidth use reduction of BW-aware over UCP+.

Figure 6.5: Relative memory bandwidth use over Static-even partitioning scheme.

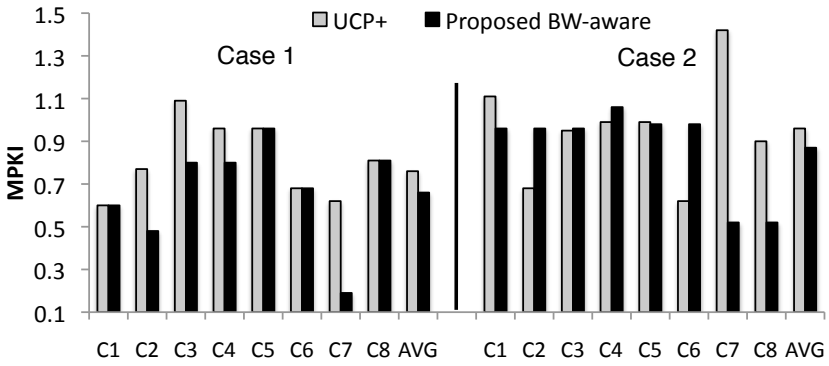
could contain the *BW-aware* algorithm within its shared memory space and therefore avoid more complex partition migrations across multiple servers. Finally, the miss rate improvements are consistent across a reasonable range of cache sizes, demonstrating a small improvement as the last-level cache size increases. The increased size allows more surplus of cache ways (see Algorithm 2) and therefore allows the *Inter-chip* algorithm to find more candidates for swapping that can potentially lead to a better mapping of the workloads to the available chips.

6.4.2 Memory Bandwidth

To show the memory bandwidth improvements we focus on the reduction of the average worst-case chip memory bandwidth in the system during an epoch. As the proposed algorithm migrates bandwidth from the highly to lightly loaded chips in the system, an average reduction of bandwidth across all the chips is not relevant in determining the performance. Instead we care for the cases that chips feature long memory latencies due to overcommitted memory bandwidth requirements. Fig. 6.5(a) includes the bandwidth reductions achieved by using *UCP+* and the proposed scheme over the static-even partitions case. The *UCP+* algorithm reductions are only due to the reductions in the miss rate achieved by the Marginal-Utility-based algorithm using the MSA-profilers information and on average is equal to 19% over the simple static case with even partitions. The addition of the *Inter-chip* algorithm in the proposed *BW-aware* scheme increases this reduction to an overall 36%. The additional gains that the migration-based scheme can provide over the *UCP+* are shown in more details in the table of Fig. 6.5(b). Overall, the scheme is able to provide an additional 18% reduction in bandwidth over *UCP+*. As expected, the bandwidth reduction is greater in the smaller cache configurations. Essentially, the higher miss rates due to smaller caches results in greater contention which provides greater opportunities for improvement. In addition, as the number of chips in the system increases, we are able to find more opportunities for migrations as such the memory bandwidth reductions increase significantly from 2 to 16 chips. The system size required for good optimizations is comparable to that required by the cache capacity optimizations, that is 8 to 16 chips. One artifact of using average worst-case bandwidth is the upward trend of the *UCP+*. As the number of chips in the simulation increases, the expected worst-case bandwidth increases due to the random nature of the workload selection.



(a) Instructions per cycle (IPC) comparisons.



(b) Misses per kilo-instructions (MPKI) comparisons.

Figure 6.6: Throughput and misses per kilo-instructions (MPKI) evaluation of two case studies.

6.4.3 Detailed Simulation – Multiprogrammed Case Study

We chose a set of experiments of the previous simulations to validate the effectiveness on a full system simulation based on Simics [48] and GEMS [49]. We simulated an 8-chip system with each chip being an 8-core CMP. We selected the experiments randomly to validate the proposed scheme and see its effectiveness on overall performance. We chose two sets of 64 random assignments and run two experiments on these sets. The first experiment uses only the *UCP+* approach which is equivalent to

the *Intra-chip partitioning* algorithm. For the second experiment we apply the multi-chip *Inter-chip partitioning* algorithm (*BW-aware*) on each set in addition to *UCP+*.

Figs. 6.6(a) and 6.6(b) show the IPC and MPKI (misses per kilo instructions) comparison of *UCP+* and *BW-aware* for each chip for the first and second experiment sets, respectively. The first set demonstrated an average improvement of 8.6% in IPC and 15.3% in MPKI across all chips, mainly due to bandwidth improvements on Chip 4 and Chip 7. The main difference between the assignments of Chip 4 and Chip 7 for the cases of *UCP+* and *BW-aware* was moving *bwaves* and *mcf* and replacing them with *povray* and *calculix*. Both *bwaves* and *mcf* are, according to the MSA-based profiling, two of the most demanding memory bandwidth benchmarks in the whole suite and compared to them *povray* and *calculix* have very low bandwidth requirements.

The second set of experiments showed an average improvement of 8.5% in IPC and 11% in MPKI. The main contributor of this reduction are the workload migrations on Chip 7. Chip 7 was initially over-utilizing the memory bandwidth creating a significant number of misses. By applying the *Inter-chip* algorithm, *bwaves* and *gcc* workloads where swapped with *zeusmp* from Chip 2 and *gamess* from Chip 6. Both *zeusmp* and *gamess* benchmarks need a small percentage of memory bandwidth while *bwaves* and *gcc* have both high memory bandwidth demands. Therefore, such migrations enabled both the high-demanded benchmarks to use the previously available bandwidth in Chip 2 and 6, and the previously saturated Chip 7, to allocate the capacity and bandwidth to the rest of its workloads. Both of these experiments validate the effectiveness of the algorithm in improving both the cache miss rate and memory bandwidth requirements in over-commited systems, and show IPC and MPKI improvements close to the one estimated in the previous section using the Monte-carlo approach.

6.4.4 Detailed Simulation – Multithreaded Case Study

As for the previous case of multi-programmed workload we simulated an 8-chip system with each chip using an 8-core CMP, that is 64 overall threads. There are two kinds of multithread workload execution styles: *homogeneous* and *heterogeneous* multithreaded. Homogeneous multithreaded workloads typically consist of heavily multithreaded workloads like SPLASH or SPECjbb2005 that create multiple identical threads working on different data sets or performing similar operations like transactions. For this kind of execution, an efficient memory-subsystem resource management has to be able to minimize the resource demand interference between the identical threads and provide a platform to allow a fair sharing of the resources among these identical threads.

On the other hand, heterogeneous multithread workloads are usually formed from dissimilar multithreaded applications that feature divergent resource demands and resource occupancy patterns. A simplified example could be an Apache server running on a portion of the system while SPECjbb transactions are served on the rest of the system. In these cases, the resource management scheme should be able to identify which subset of threads can benefit more from each resource and assign them accordingly aiming for overall system performance and/or fairness improvement. Heterogeneous workloads are more challenging in resource managing as a thread unaware scheme might favor some of the threads from one applications but not all. This approach can be really wasteful in a real system as multithreaded applications are typically synchronized with the use of software and hardware barriers. Improving the performance only of a subset of an applications threads might improve their performance, helping them to reach a barrier faster. This performance improvement is wasted though if they have to wait idle in the barrier in order to synchronize with the rest of the applications' threads.

To evaluate the presented scheme we simulated experiments for both cases. First we simulated four of the SPLASH-2 benchmark (the ones with the most divergent

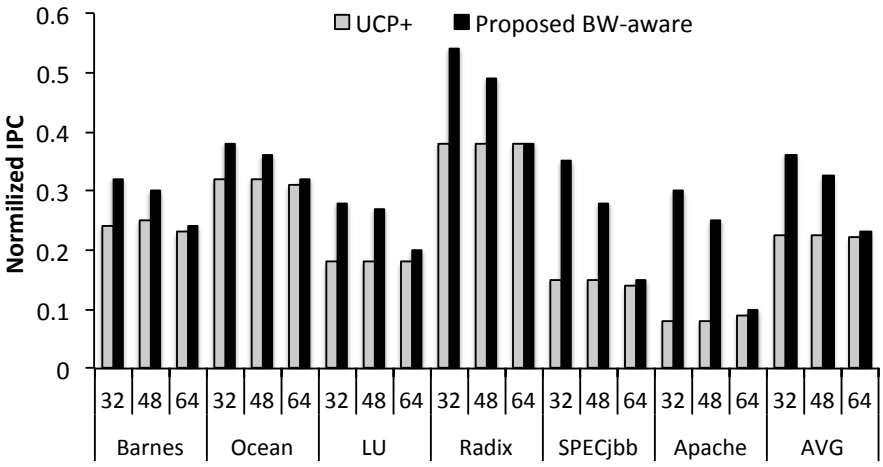


Figure 6.7: Relative IPC improvement of homogeneous multithreaded workloads over static-even partitions on each chip for 32/48/64 concurrent executing threads.

behavior), SPECjbb2005 and Apache homogeneous workloads on the whole system using one workload per time and scale them to from 32 to 64 threads. For the cases of less than 64 threads we use Solaris default thread scheduler that sequentially fills up cores with jobs. Fig. 6.7 shows the results of our experiments relative to using static, even partitions on each chip. For every case we estimate the IPC only for the active threads. When the machine is full with 64 threads there is little to do over the UCP+ since all threads have similar requirements and both schemes optimize the resource allocation on each chip. As there is small difference between the workloads executed on each chip our bandwidth-aware proposal does not have big opportunities in improving performance across the whole system. On the other had, when less than 64 threads are used and since we assigned threads to cores using Solaris sequential scheduler the bandwidth-aware scheme could amortize the bandwidth use across multiple chips instead of having a small number of chips full of threads and other chips empty. This is behavior is more evident for the case

Table 6.2: Heterogeneous Multithreaded Workloads.

Set	Benchmarks [benchmark(#threads)]
1	Barnes (32) and SPECjbb2005 (32)
2	Ocean (32) and SPECjbb2005 (32)
3	Barnes (32) and APACHE-2 (32)
4	Ocean (32) and APACHE-2 (32)
5	Barnes (16), Ocean (16), LU(16), Radix(16)
6	SPECjbb2005 (32) and APACHE-2 (32)

of SPECjbb and Apache2 were moving out heavy threads from a chip helped up to 30% over static-even partitions while UPC+ could provide a small performance improvement close to 6% only. Overall our approach managed to move thread from high utilized chip to chips with lower utilizations. Of course such results could come from a more sophisticated, load-balancing, scheduler that can distribute the jobs across the multiple threads. Nonetheless, these results provide an indication that our scheme can effectively swap applications threads and provide a better use of the on-chip resources and memory bandwidth.

Fig. 6.8 shows our results of combining multithreaded workload to create heterogeneous workloads. Table 6.2 shows the combination of workloads that we assumed in our scheme. Unfortunately, there is huge space of forming experiments for the case of heterogeneous workloads that one could modify the mix of workloads and the number of threads assign to each one. As in this study we just want to analyze the behavior of the proposed scheme using multithreaded workloads but we do not specifically target them in our scheme, we decided to include a small representative combination of workloads in our sets. Again as for the previous case of homogeneous multithreaded workloads, we use Solaris 10 scheduler that sequentially assigns threads to cores. From the results we can see

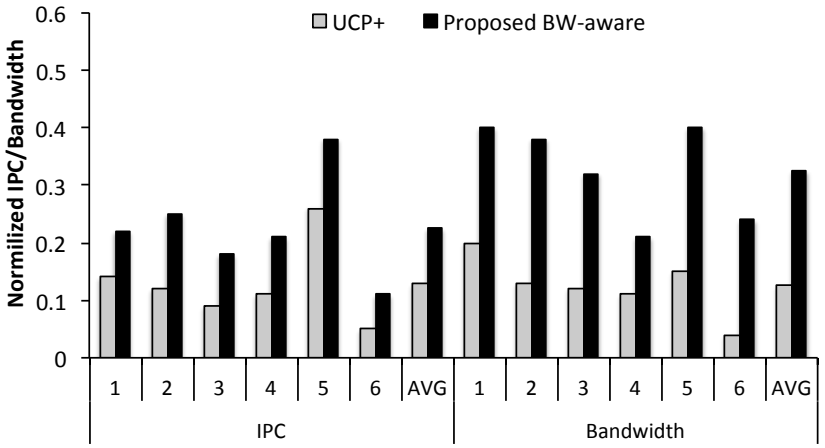


Figure 6.8: Relative IPC and bandwidth use improvements of heterogeneous multithreaded workloads over static-even partitions on each chip for 32/48/64 concurrent executing threads.

that our scheme can effectively improve the performance and bandwidth use by finding the best combination of threads that should be run on each chip. By swapping applications between cores, the proposed scheme move demanding threads that were assigned on the same chip to chips with less demanding threads (from another workload). On average the proposed scheme can find an assignment per chip that includes a good combination of threads from every application, to allow better sharing of the on-chip cache and reduce overall system memory bandwidth. The proposed scheme is especially effective in the cases 5 and 6 were multiple diveregent workloads execute on the system. For the case of 5 it is able to find the correct mix of the four different threads to run on each chip while in the case of 6 the improvement coming from avoiding overcommitting in memory bandwidth chips. Apache2 threads are more demanding in memory bandwidth but SpecJBB threads need more cache resources to process their transactions. As a result, our scheme is able to combine those two threads in order to provide a more efficient use of the on-chip cache

while keeping a lower contention on the external bandwidth required per chip. Such combination of threads provide significant improvement over UCP+, with up to 30% better bandwidth utilization and close to 28% IPC improvement. Finally, the cases of heterogeneous multithreaded workloads can be further improved if our algorithm is aware of the application that each thread belongs to. Such knowledge would allow it to treat each application's threads similarly by grouping them together in thread sets, rather than trying to find opportunities for every individual thread in the system. Such approach is left for our future work.

6.5 Related Work

Kim *et al.* [38] highlighted the importance of enforcing fairness in CMP caching and proposed a set of fairness metrics for optimization. However, their policies are analyzed using only best static offline parameters and mechanisms. Suh *et al.* [77] propose a dynamic L2 cache partitioning technique that allocates the L2 cache resource at the granularity of cache ways to improve system throughput. They employ a greedy heuristic to allocate cache ways to the application that has the highest incremental benefit from the assignment. Qureshi *et al.* [63] improves Suh's allocation policy by using cache utility monitor (UMON) to estimate the utility of assigning additional cache ways to an application. Both of them target at improving the throughput of a single CMP system through reductions in cache misses. Chang *et al.* [7] proposed time-sharing cache partitions, which translates the problem of fairness to the problem of scheduling in a time-sharing system. Jiang *et al.* [29] demonstrate that co-scheduling on k-core ($k > 2$) is NP-hard, and propose a greedy algorithm that schedules jobs in the order of their sensitivity to cache contentions. The proposed approach in this chapter extends these papers by proposing a systematic approach that considers main memory traffic in addition to cache miss rate. In addition we seek system-level solutions beyond a single chip in

the context of large CMP systems. Iyer *et al.* [23] proposed a framework for enforcing QoS characteristics in a system based on trial-and-error approaches to fit the QoS targets. That work was later on extended by Zhao *et al.* [88] with a set of counters, named CacheScouts, that monitored their QoS characteristics in a system and made resource management decisions accordingly. We use non-invasive MSA profilers that can predict resource requirements for all different cache capacity assignments in parallel for both misses and bandwidth use. Such prediction is important for reconfigurable schemes in order to perform expensive reconfiguration operations only when the benefit is predicted to be important enough.

6.6 Summary

Shared resource contention in CMP platforms has been identified as a key performance bottleneck that is expected to become worse as the number of cores on a chip continues to scale to higher numbers. According to the provided analysis, bandwidth-aware optimizations at the system level provide significant improvements over optimizations limited to a single chip with very small additional hardware overhead. The proposed *Bandwidth-aware* partitioning scheme is able to provide on average, 25.7% reduction in misses and 36% reduction in worst-case bandwidth use compared to static-even last-level cache partitioning schemes. A comparison with a state-of-art cache partitioning scheme showed an average 18% reduction in memory bandwidth along with 7.9% reduction in the last-level cache miss rate. In addition, detailed simulation case studies show that the proposed resource management scheme can achieve an average reduction of 8.5% in IPC and 13% in MPKI on a 8-chip CMP system. These improvements over existing techniques can justify the hardware overhead of the proposed MSA-based profilers which was estimated to be 1.4% of the baseline 8MB last-level cache.

Chapter 7

High Performance Quasi-partitioned Last-level Caches

To achieve high efficiency and prevent destructive interference among multiple divergent workloads, the last-level cache of Chip Multiprocessors has to be carefully managed. The proper manipulation of last-level CMP caches is of primary concern, as it significantly contributes to the overall throughput and power consumption. Previously proposed cache management schemes suffer from inefficient cache capacity utilization, by either focusing on improving the absolute number of cache misses or by allocating cache capacity without taking into consideration the applications' memory sharing characteristics. Reduction of the overall number of misses does not always correlate with higher performance, as Memory-level Parallelism can hide the latency penalty of a significant number of misses in out-of-order execution. This chapter presents a quasi-partitioning scheme for last-level caches, MCFQ, that combines the memory-level parallelism, cache friendliness and interference sensitivity of competing applications, to efficiently manage the shared cache capacity. The proposed scheme improves both system throughput and performance fairness – outperforming previous schemes that are oblivious to applications' memory behavior. Using full-system simulations, the MCFQ on a 4-core CMP achieved an average improvement of 10% in throughput and 9% in fairness over the next best scheme, with some specific cases reaching an improvement of up to 18% and 23% in throughput and fairness, respectively.

7.1 Introduction

To address the problem of contention in shared last-level caches, three main research directions have been proposed in the past: a) *Cache Capacity Partitioning* schemes [6, 14, 26, 38, 63, 86], b) *Cache-blocks Dead-time* management schemes [26, 61], and c) *Cache Pseudo-partitioning* schemes [86]. The *Cache Partitioning* schemes identify an ideal size of capacity that each concurrently executing thread should be assigned to maximize throughput and/or fairness. Although such schemes can effectively eliminate threads' destructive interference by utilizing isolated partitions, they result in inefficient capacity utilization [26, 86]. *Cache-blocks Dead-time* management schemes focus on identifying the dead (not any more useful) cache lines and force their early eviction from the cache. As a result, more useful lines are retained in the cache, enabling better utilization of capacity. Unfortunately, since there is no control over the number of cache lines each thread can maintain in the cache, destructive interference is still present. To counter this problem, *Cache Pseudo-partitioning* schemes [86] have been proposed that provide a combination of both previous schemes; they allow the applications to share part of the available capacity and compete for it, while, based on their cache space demand rate and their ideal cache partition capacity size, can occupy additional capacity from other applications to keep useful lines in the cache. An effective cache management scheme should provide both good *capacity utilization* and *interference isolation*. While both Dead-time and pseudo-partitioning schemes identify and prevent thrashing benchmarks from polluting the cache, they are oblivious to the other two important memory behaviors that we identify in this paper, namely, *Cache Friendly* and *Cache Fitting* behavior. As the evaluation section shows, a scheme that is aware of application's Cache Friendliness (memory behavior) can provide better capacity utilization while significantly reducing cache interference.

Previous cache management schemes allocate capacity to applications based on

heuristics that aim at the reduction of the absolute number of cache misses. In out-of-order execution, different workloads may have a different number of overlapping cache misses (a.k.a Memory-level Parallelism - MLP) which could drastically change the workload's performance sensitivity to the absolute number of misses. As this chapter will show, targeting the reduction of absolute number of misses introduces significant inefficiencies. A significant percentage of cache space can be allocated to applications with high cache demand rate that cannot actually extract any benefit from the dedicated capacity; while low MLP, miss-latency sensitive workloads with small demand rates can suffocate in small cache partitions. As the number of cores per die is increasing, such wasteful cache management schemes will severely affect performance.

This chapter presents a *Memory-level parallelism and Cache-Friendliness aware Quasi-partitioning scheme* (MCFQ). The scheme estimates the sizes of the overlapping cache partitions for competing applications by taking applications' MLP into consideration. To minimize the effects of interference among different applications, this research utilizes application's *Interference Sensitivity factor* for assigning the MCFQ priorities and insertion points. Finally, the scheme scales the quasi-partitions sizes, to ensure that the average occupancy of cache ways is close to the ideal ones that need to be enforced on the last-level cache. It is named *Quasi-partitioning* because it mimics the operation of capacity partitioning schemes without actually enforcing the use of isolated partition; while at the same time provides the benefits of pseudo-partitioned schemes. Overall, MCFQ makes the following contributions:

1. Proposes the use of Memory-level Parallelism (MLP) information in the cache partitioning scheme to predict applications' final performance sensitivity to last-level cache misses. Such scheme targets overall system throughput improvements instead of raw reductions of cache misses that previous proposals did.

2. Describes a priority scheme that assigns priorities to applications based on their

"Cache-friendliness". *Friendly, Fitting and Thrashing* applications get different priorities that are controlled through the cache line insertion points in the last-level cache.

3. Utilizes a "Interference Sensitivity" factor to identify how much sensitive is the behavior of an application to last-level cache contention and how much it is expected to hurt the cache behavior of other applications.

4. Proposes a "Partitions Scaling" scheme to dynamically adjust the quasi-partition sizes in order to match the average cache-ways occupancy of each application with the ideal estimated ones.

5. Finally, the proposed MCFQ policies combines all of the previous contributions in a single scheme which provides significant improvements over previously proposed representative schemes. On a 4-core CMP system MCFQ achieved improvements of up to 18% in throughput (10% average) and 23% fairness (9% average) over the next best scheme.

7.2 Motivation

7.2.1 Memory-level Parallelism (MLP)

In out-of-order execution, Memory-level Parallelism (MLP) can drastically change workload's performance sensitivity to the number of last-level cache misses. MLP causes the cache miss events to be clustered together, overlapping their miss latency. The effective performance penalty of each last-level cache miss can vary according to the application's *concurrency factor* (MLP) and in general is expected to be smaller than the main memory access latency of a typical L2 miss. The two extreme cases of MLP are pointer chasing code and vector code. Pointer chasing code has a *concurrency factor* of 1, since all of its loads are dependent on previous loads and therefore only one memory access can take

place at the same time. In this case, reducing the number of last-level cache misses directly translates into performance gains. On the other hand, vector code has a large concurrency factor that is determined by the maximum number of outstanding misses that the micro-architecture can support. In vector code, many independent load misses can be overlapped with each other, forming clusters of misses. A raw reduction of these overlapping misses may not lead to noticeable performance gains. Each miss cluster introduces a constant latency to reach main-memory and one have to avoid the whole cluster to improve performance. As a result, any cache managing scheme that just aims at the reduction of absolute number of cache misses, introduces inefficiencies when targeting overall throughput optimization and cannot provide any safe QoS guarantees.

7.2.2 Cache Friendliness Aware Cache Allocation

Prior work has demonstrated that the capacity of the last-level cache has to be carefully managed in order to prevent destructive interference among multiple divergent workloads [6, 14, 26, 63, 86]. When multiple threads compete for a shared cache, cache capacity should be allocated to the threads that can more efficiently use such resources [20, 63, 76]. The commonly used LRU replacement policy is unable to identify which applications can effectively use the extra capacity and which ones act as cache hogs, polluting the cache. LRU assumes all cache misses to be equally important and allocates the capacity based on the cache demand rate. Therefore, under LRU policy, applications with high demand that cannot efficiently use the extra capacity, receive resources that could otherwise have been used by the applications that can benefit from it. *Cache capacity partitioning* schemes address this problem by allocating specific, isolated, cache partitions to each application, eliminating the potential destructive interference. In these cases, efficient cache management algorithms are necessary to identify the ideal cache partition size for each application. Such algorithms have to allocate enough capacity to

each concurrently executing application to be able to fit its working set.

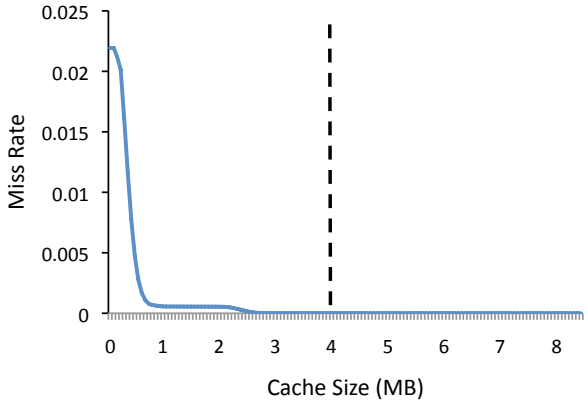
7.2.2.1 Need for Cache Pseudo-partitioning Schemes

Despite their wide spread, it has been shown [26, 86] that isolated cache partitioning schemes tend to be wasteful since, some percentage of dedicated cache capacity may be underutilized. To solve this inefficiency, *Cache pseudo-partitioning* schemes have been proposed [86] that enforce an average partition size for each application. To do so, they break the cache partitions' isolation assumption and allow applications to share cache capacity and therefore compete with other applications for it. Such approaches provide the flexibility to the cache management scheme to "steal" cache capacity from cache-underutilizing applications and temporarily allocate it to the applications that can benefit the most from the extra capacity. Of course, such flexibility is not coming for free. By breaking the partitions' isolation assumption, destructive interference between the cache capacity competing cores is introduced. It is therefore necessary to allocate resources based on both the performance sensitivity of each application to cache space and how well the cache demands of each application compete with each other.

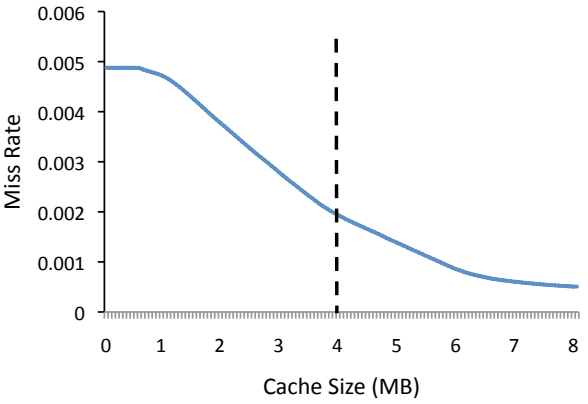
7.2.2.2 Applications Cache Behavior

To better understand the cache demands of typical applications, Fig. 7.1 illustrates the miss rate of different classes of applications, as the cache capacity allocated to them changes. Based on an analysis done on a typical CMP with 4MB of shared last-level cache, the applications can be classified into three basic categories:

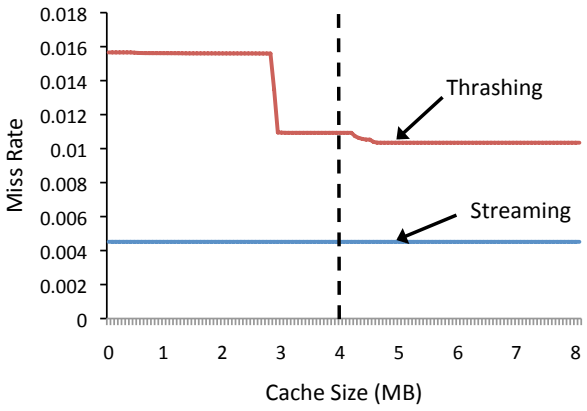
1. **Cache Fitting:** These applications have a small working set size which can easily fit in a typically sized shared cache, but at the same time, they are also very sensitive to the allocated cache size. As Fig. 7.1(a) shows, an allocation which is slightly smaller than their ideal size can significantly increase their miss rate, forcing them to behave as



(a) Cache Fitting Application (dealII).



(b) Cache Friendly Application (gcc).



(c) Cache Thrashing / Streaming (lbm, milc).

Figure 7.1: Three representative categories of miss-rate sensitivity of applications to cache space.

cache thrashing applications. It is, therefore, important for a shared cache management scheme to allocate enough space to these applications to fit their working sets. If such cache space allocation cannot be guaranteed then it is preferable to reduce the cache resources allocated to them in order to restrict their thrashing behavior from affecting the rest of the applications.

2. **Cache Friendly:** These applications can in general benefit from additional cache space, as shown in Fig. 7.1(b). These applications can efficiently share the cache capacity using an LRU-like replacement policy. Under a *Pseudo-partitioning* scheme, these applications should have the higher priority since they have the potential to benefit from any additional capacity allocated to them based on their cache space demand rate.

3. **Cache Thrashing/Streaming:** These applications feature a poor cache locality due to their big working set size that cannot fit in a typically sized shared cache. As a result, under LRU replacement policy, they pollute the shared cache without actually getting any benefit from the occupied cache capacity. When they execute with cache friendly or cache fitting applications, they should be assigned the smaller priority over the other types of applications, allocating to them the minimum cache space. We will refer to both *Thrashing* and *Streaming* applications as *Cache Thrashing* since both generate a thrashing behavior on the cache.

Overall, we can extract three basic rules that a cache capacity management scheme has to follow: a) the cache resources have to be allocated proportionally to the applications' cache space benefit, b) *Thrashing* applications have to be identified and restricted to a small fraction of the cache to significantly reduce destructive interference, and c) *Fitting* applications are sensitive to the cache space allocated to them and the management scheme has to provide some space allocation guarantees to avoid overall system degradation.

This chapter presents a quasi-partitioning scheme that allocates cache capacity based on the *Cache Friendliness* of each application. For the case of *Cache Thrashing* applications, *Bimodal Insertion Policy (BIP)* [61] and later on *Thread-aware Dynamic Insertion Policy* [26] have shown that by inserting the new coming cache blocks of such applications in the LRU position, cache thrashing can be significantly avoided. Therefore, the proposed scheme allocates the lowest priority to *Cache Thrashing* applications, restricting them to the lower 1-2 cache ways of the LRU stack. The proposed scheme treats *Cache Fitting* applications with an intermediate priority level and provides some minimum space allocation guarantees, to allow them to fit in the cache and execute with low miss rate. *Cache Friendly* applications receive the highest priority since they can benefit the most from the cache space allocated to them. For the case of *Cache Friendly* and *Cache Fitting* applications, when multiple concurrently executing applications belong to the same category, the scheme allocates priorities within each category based on their *Interference Sensitivity Factor*, i.e. how friendly each application is to the co-running applications when it has to share the cache.

7.3 MCFQ: MLP and Cache-friendliness aware Quasi-partitioning Scheme

The proposed MCFQ scheme, manages the last-level cache by allocating cache quasi-partitions to competing applications based on: a) the performance sensitivity of misses (MLP-aware), b) applications' cache memory behavior (Fitting, Friendly and Thrashing), and c) their interference sensitivity factor. This section, first explains the profiling mechanisms we used to capture applications' behavior followed by the MLP-aware assignment of capacity. In the end of the section, the overall proposed MCFQ scheme is described in details along with an example of it operation.

7.3.1 Profiling Applications' Cache Demands, Average Concurrency Factor and Memory Behavior

Applications Cache Demands: The on-line cache demands monitoring scheme used in this study is based on the principles introduced by Mattson's stack distance algorithm (MSA) [50]. The proposed implementation is based on previously proposed hardware-based MSA profilers for Last-level cache misses [32, 63, 89]. To monitor each core individually, the scheme implements an individual profiler for every core which assumes that the whole cache is dedicated to it. The profilers operation is described in details in Section 4.1. Such profiler is able to provide a cache miss rate curve for each core that correlates cache misses with allocated cache capacity. The overhead of the specific profiler used in this study is estimated to be 117 kbits per profiler, which is approximately 1.1% of the size of an 4MB Last-level cache design; assuming 4 overall profilers for a 4-core CMP.

Doing so, we maintain a set of shadow cache tags per core that allows us to monitor what would be each core's cache use if the core had exclusive access to the cache. We use *N+1* counters (N=number of cache-ways) that profile the cache accesses based on the LRU stack distance where each access had a cache hit. Based on the positions of the hits in the profiler, we can create a cache miss rate curve that correlates cache misses with allocated cache capacity.

To reduce the profilers overhead we used *partial hashed tags* [35] and *set sampling* [34]. In the proposed implementation we use 11 bit *partial tags* combined with 1-in-32 *set sampling*. Such profiler added only 1.3% more misses than the performance we got using a full tag implementation, over the whole SPEC CPU2006 [16] suite. The profilers counters' size was set to 32 bits to avoid overflows and we implemented the LRU stack distance of the MSA as a single linked list with head and tail pointers. Overall, the implementation overhead is estimated to be 117 kbits per profiler, which is approximately 1.1% of the size of an 4MB Last-level cache design; assuming 4 overall profilers.

Profiler for Concurrency Factor (MLP): To estimate the average *Concurrency factor* (MLP) of each application running on a single core, we profiled the *Miss Status Holding Registers* (MSHR). The MSHR typically exists in an out-of-order core between the L1 and L2 cache [41]. The purpose of the MSHR is to keep track of all the outstanding L1 misses being serviced by the lower levels of memory hierarchy (i.e. L2 and main memory). Therefore, the number of entries in the MSHR represents the number of the concurrent, long-latency, outstanding memory requests that were sent from the core to the last-level cache or memory, which is equal to the MLP of the application running on the core. To estimate the average MLP, we modified the MSHR by adding two counters: one to hold the aggregated number of outstanding misses in the MSHR, and a second one to hold the overall number of times an L1 miss was added in it. Both counters are updated every time an entry is added (new outstanding demand miss) or removed (demand miss served) from the MSHR.

Identify Applications Cache Friendliness Type: To categorize the cache behavior of each application (Fig. 7.1), the scheme utilizes the previous cache miss profiles that the profilers create for each core. The profiler looks at the estimated Misses Per Kilo Instructions (MPKI) when only one cache-way is allocated to an application ($MPKI_{min.capacity}$), and the MPKI when the whole cache capacity ($MPKI_{max.capacity}$) is given to it. To assume an application as $Thrashing$ one the following has to be true:

$$\frac{MPKI_{max.capacity}}{MPKI_{min.capacity}} > Threshold_{Thrashing} \tag{7.1}$$

In the same way, if at any capacity point of the cache miss profile (number of cache-ways dedicated to the core), is estimated smaller than a $Threshold_{Fitting}$ the application is characterized as $Fitting$. The remaining applications are set to $Friendly$.

Both thresholds ($Threshold_{Thrashing}$ and $Threshold_{Fitting}$) are system parameters which can be tuned to the specific system and the level of aggressiveness that is required by the partitioning scheme. For the analysis presented in this chapter, we performed a sensitivity analysis for both parameters to find the best values that best separate thrashing/streaming and fitting applications based on their cache miss profiles. Our analysis, on SPEC CPU2006 benchmarks, showed that a threshold of $Threshold_{Thrashing} = 0.86$ provides the best trade-off for characterizing applications as thrashing/streaming. Bigger values allows applications with thrashing behavior to be characterized as friendly and pollute the cache. On the other hand smaller values of this threshold treated potentially friendly applications that could be benefitted from additional cache capacity (even marginally) as thrashing applications, restricting them in one or two LRU ways in the cache. Finally the best value for the $Threshold_{Fitting}$ value was found to be 0.005 of the $MPKI_{min.capacity}$. This value was enough to guarantee that when an application achieves less than 0.5% of its initial cache miss rate at any cache allocation dedicated to it, we can treat it as a fitting application.

7.3.2 Allocation of Cache Capacity based on Performance Sensitivity of Misses

As discussed before, an efficient last-level cache managing scheme must take into consideration the effects of multiple outstanding misses to performance and execution fairness. To model this sensitivity, we separate the execution time spent for each application into two parts: the time spent inside the core ($T_{Perfect\_Cache}$), assuming a prefect cache, and the time consumed outside the core serving cache misses (T_{Misses}). From the two, only T_{Misses} is subject to the interference due to: a) the resource sharing of the last-level cache capacity, and b) interconnection and memory controller bandwidth sharing, with other concurrently executing applications. Overall, to model application's performance in terms of *Cycles-per-Instruction* (CPI) we can use Equation 7.2 [69].

Assuming a single L2 acting as the last-level cache, the CPI_{Misses} can be broken down as the CPI when we hit in L2 (CPI_{Hit_L2}) and the delta of CPI due to L2 misses that are directed to the main memory (CPI_{Memory}).

$$CPI = CPI_{Perfect\_Cache} + CPI_{L1\_Misses} = CPI_{Perfect\_Cache} + CPI_{Hit\_L2} + CPI_{Memory} \quad (7.2)$$

Equation 7.2 though does not account for any concurrency among cache misses outside a core, that is MLP (average number of *useful* outstanding requests to the memory [8]). Therefore, we can define CPI_{Memory} as:

$$CPI_{Memory} = \sum_{i=0}^{N_{misses}} \left(\frac{Miss\_Penalty_i}{Memory\_reference} \times \frac{Memory\_references}{Instructions_i} \right) \quad (7.3)$$

$$= Miss\_Rate_{memory} \times \frac{Miss_{Latency}}{Concurrency\ Factor} \quad (7.4)$$

Since only the CPI_{Memory} term of Equation 7.2 is affected by last-level misses, Equation 7.4 is a direct way to estimate the impact of last-level cache misses to final performance. As $Miss_{Latency}$ can be approximated to be equal to the average effective latency of an L2 miss to the main memory, the only information missing form the equation is the $Miss\_Rate_{memory}$ and the *Concurrency Factor* (MLP) of an application. To estimate both terms, we utilize the online profiling mechanisms; described in the previous section.

7.3.2.1 Cache Capacity Allocation Algorithm

The cache capacity assignment scheme used in this reseach is based on the concept of *Marginal Utility* [81]. This concept originates from economic theory, and has been used

in cases where a given amount of resources (in the case cache capacity) provides a certain amount of *Utility* (reduced misses or CPI). The amount of *Utility* (benefit) relative to the used resource is defined as the *Marginal Utility*. Specifically, the *Marginal Utility* for n additional elements when c of them have already been used is defined as:

$$Marg.\,Utility(n) = \frac{Utility\,Rate(c+n) - Utility\,Rate(c)}{n} \tag{7.5}$$

The cache miss profilers (Section 7.3.1) provide a direct way to compute the *Marginal Utility* for a given workload across a range of possible cache allocations. We follow an iterative approach, where at any point we can compare the *Marginal Utility* of all possible allocations of unused capacity. Of these possible allocations, the maximum *Marginal Utility* represents the *best* use of an increment in assigned capacity. The greedy algorithm implementation used in this chapter follows the one initially introduced by Stone *et al.* [73]. The algorithm estimates the cache capacity partitions assuming that each partition is isolated from each other. These partitions will be the ideal partition sizes that we want to enforce on the shared cache.

To take MLP into consideration, the marginal utility is estimated based on the cumulative histograms of the last-level misses and the application's concurrency factor. The *Utility* function of the algorithm represents the effective additional benefit that each cache way can contribute to the final performance of an application. The *Utility Rate* function, is estimated as the fraction of the cumulative hits at each cache-way over the average concurrency factor of the core. That is:

$$Utility\_Rate(cache\ way) = \frac{MSA\_hits(cache\ way)}{Average\_Concurrency\_factor} \tag{7.6}$$

The $CPI_{Perfect\_Cache}$ of Equation 7.2 is expected to remain unchanged for the same application over a small, steady execution phase (10M instructions in this paper). Therefore, by modifying the cache allocation of each core, we affect the other two terms in the equation, that is CPI_{Hit_L2} and CPI_{Memory}. Following this approach, the algorithm can effectively allocate capacity to the cores that affect the final observable CPI the most.

7.3.3 Cache-friendliness aware Quasi-Partitioning Scheme

7.3.3.1 Cache-line Insertion, Promotion and Replacement Policy

Using the terminology from [86], a cache management scheme requires three basic pieces of information for its operation: a) an *Insertion Policy*; the location (cache-way) where to insert a new cache line in the LRU stack per application/core, named IP_i, b) a *Promotion policy*; the way LRU stack is modified on a cache hit, and c) a *Replacement Policy*: the exit point of cache-lines in case of line eviction. A cache partitioning scheme like UCP [63] features one isolated partition per core that occupies a number of specific cache-ways, equal to its partition size S_i. The partitions are non-overlapping so $\sum S_i = \#(cache\_ways)$. Using the above terminology, the insertion point of each core is set to the top of each partition; the promotion policy is set to move the cache line that got a hit to the MRU position of its partition; and the replacement policy is LRU among the ways that belong to the same partition. On the other hand, a pseudo-partitioning scheme like PIPP [86] does not have isolated partitions like UCP. All the cache ways are accessible by all the cores, simulating a typical LRU stack. To allow such pseudo-partitions, PIPP sets the insertion point (IP_i) of each core to be at each core's partition size, $IP_i = S_i$ (LRU position is size 1 and MRU position is size 16 for a 16-way cache); the promotion policy was set to promote cache hit lines by one position based on a P_{prom} probability (otherwise stays unchanged); and the replacement policy was set to LRU evicting from the LRU position of the shared stack. Based on this scheme, there is a shared/unallocated portion

of capacity from the largest IP_i to the MRU. This space can be utilized by applications' hot cache lines through line promotions; thus helping the cores with good cache locality to retain data in the cache.

The target of the proposed MCFQ scheme is to provide the cache-capacity utilization benefits of pseudo-partitioning schemes while minimizing the potential inter-application interference from sharing such capacity. As the evaluation results show, previous pseudo-partition schemes feature a significant level of interference and therefore a performance drop when executing high resource demanding applications. Such drop is mainly due to their Insertion/Promotion polices. The provided analysis showed that most of the applications could not efficiently utilize the extra capacity available in the cache from MRU position to the largest IP point. The benefits from using this capacity were not enough to overcome the benefits of actually allocating this capacity to specific threads, like isolated partitions do. Moreover, the interference that was introduced in the lower part of the cache is quite significant, especially for the Cache Fitting applications with small IPs.

MCFQ Policies: To overcome these problems, MCFQ estimates the IPs on the basis of a) MLP and ideal partition sizes estimated from the *Marginal-Utility* algorithm (Section 7.3.2.1), b) applications' cache friendliness, and c) Interference sensitivity factors (Section 7.3.3.2). The promotion policy is modified such that a cache line moves to its IP on a hit, i.e, a cache line is never allowed to move beyond its IP. Assuming an LRU replacement policy, the selected values for IPs can be translated in priorities P_i, with higher value for IP meaning higher priority. The analysis in Section 7.2.2 showed, we should allocate the higher priority to *Cache Friendly* applications followed by *Cache Fitting* ones. We restrict *Cache Thrashing* applications at the lowest 1 cache way of the LRU stack, similar to the TADIP [26] and PIPP [86] schemes. Thus the applications with higher

priority have less competition in terms of promotion and demotion of cache lines in the LRU stack. The *Partition Scaling* scheme (Section 7.3.3.2) scales the estimated partition sizes and IPs to ensure that *Cache Fitting* applications have enough capacity to avoid thrashing the cache. Between applications of the same category, we allocate priorities based on applications' interference sensitivity factor described in 7.3.3.2. Such approach significantly reduces the cache interference for low priority cache-ways. Since we share lower priority partitions, the *Partition Scaling* scheme makes sure that each application has enough capacity to maintain, on average, a number of cache ways close to the ideal estimated partition size. Therefore, for this case under study $\sum S_i > \#cache\_ways$. By doing so, we can allow *capacity stealing* for high priority cores while maintaining a low threshold of capacity allocation to the lower-priority partitions.

7.3.3.2 Interference Sensitivity Factor and Partition Scaling Scheme

Interference Sensitivity Factor: While MCFQ has a clear priority scheme between applications with different cache friendliness behavior, we need a way to allocate individual priorities when more than one applications belong to the same category. We base the ordering on how sensitive is an application to cache contention and how much we expect it to hurt the cache behavior of other applications. To calculate the sensitivity of an application, we used the stack-distance profiles from the cache miss profilers (Section 7.3.1) to estimate the following *Interference Sensitivity Factor*:

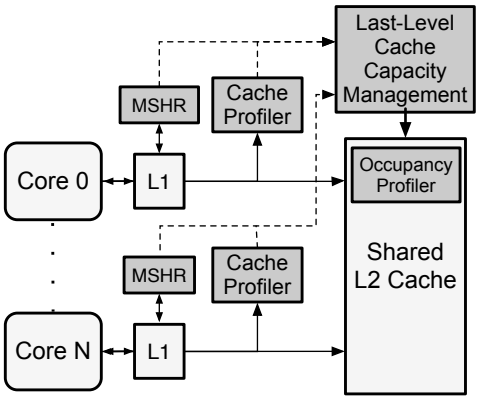
$$Interference\ Sensitivity\ Factor = \sum_{i=0}^{\#ways-1} i * hits(i) \tag{7.7}$$

$Hits(i)$ is the actual number of hits from the profiler on the i-th position in the stack, where $i = 0$ is the MRU position and $i = \#ways - 1$ is the LRU position. The more hits an application has at cache-ways closer to LRU, the more sensitive is the application to

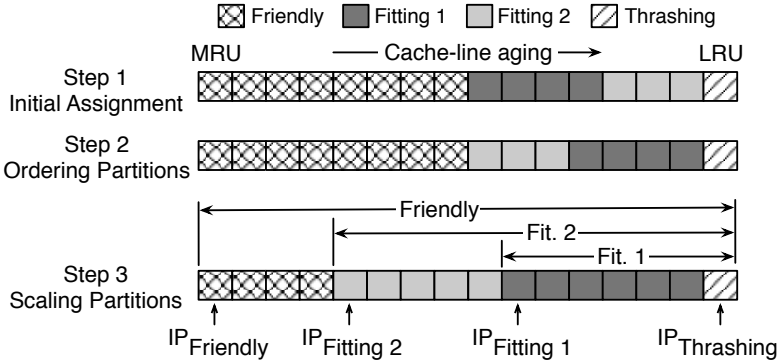
the cache contention. Under high contention, the effect of interference in misses is, on average, equivalent to allocating less cache space since a core's lines get evicted with a rate higher than its own demand rate. Hits closer to LRU positions have higher probability to become misses when the cache is shared. Moreover, in a quasi-partitioning scheme where applications insert cache lines in different insertion points, the more hits an application has in LRU positions, the more useful cache lines will be evicted by other threads that insert lines at lower insertion points. Therefore, an application with high *Interference Sensitivity Factor* is more likely to see a degradation in performance.

Partition Scaling Scheme: Since we selectively share a portion of the cache with multiple threads, we need to know what relative percentage of its allocated cache ways (based on its priority) an application actually maintains in the cache. If this number is significantly smaller than its ideal partition size, the application will feature a significant performance degradation.

To do so, we added a monitoring scheme that utilizes the cache's core inclusivity bits to measure the average number of cache-ways per core. Overall, we added two counters for every core the system supports, the $Occupancy\_Counter_i$ and $Cache\_Accesses\_Counter_i$. Whenever there is a cache hit or insertion of a new cache-line triggered by $Core_i$ we update the $Occupancy\_Counter_i$ with the number of cache ways the $Core_i$ occupies in the set, and increase $Cache\_Accesses\_Counter_i$ by one. Having this information, we can estimate the average number of ways a core occupies. If the average occupancy is less than 80% of the ideal size estimated by the *Marginal-Algorithm*, we increase the IP_i proportionally in the next epoch (see next section for epochs) to bring the occupancy close to the ideal. This is especially important for *Cache Fitting* applications to avoid them thrashing the cache. If the correction is not feasible, we set the application to insert in the MRU position. The value of 80% percent was selected after a sensitivity



(a) Overall MCFQ scheme.



(b) MCFQ operation with four applications on a 16-way cache.

Figure 7.2: Overall MCFQ scheme on a typical CMP system and an example of allocation on a 16-way cache.

analysis. More than 80% the scheme was too aggressive on scaling the partitions while a smaller threshold was not able to provide significant benefits. Notice that, since we use the average occupancy numbers, the cores can still "steal" useful capacity in a subset of cache-sets.

7.3.4 Overall dynamic Last-level cache Quasi-partitioning scheme

Fig. 7.2 illustrates the proposed last-level cache Quasi-partitioning scheme along with a simplified example. The dark shaded modules in Fig. 7.2(a), indicate the additions/modifications over a typical CMP system. Each core has a dedicated *Cache Profiling* circuit (Section 7.3.1) and we augmented the L2 design to add the *Cache-way Occupancy* profiler (Section 7.3.3.2). These two profilers along with the MLP statistics that we get from the MSHR (Section 7.3.1) are the inputs to the *Cache Capacity Management* module that controls the last-level cache. The overall proposed dynamic scheme is based on the notion of epochs. During an epoch, the profilers constantly monitor the behavior of each core. When an epoch ends, the profiled data of each core is passed to the *Marginal-Utility* algorithm to find the ideal cache capacity assignment based on MLP. In parallel, the profilers identify the memory behavior that each application fits (Friendly, Fitting, Thrashing) and based on their category and their cache capacity assignment S_i, we decide the insertion points, IP_i, for each core (Section 7.3.3.1). If there are more than one applications in the same category, the *Interference Sensitivity Factor* is used to assign the priority among them. Finally, the *Partition Scaling Scheme* is used to scale the estimated partitions sizes and therefore their insertion points to ensure that the scheme can on average meet the ideal partition sizes estimated by the *Marginal-Utility* algorithm. The whole process is repeated at the end of the next epoch. In the evaluation section we used epochs of 10M instructions.

Fig. 7.2(b) illustrates an example of how MCFQ operates on a 4-core CMP using a 16-way last-level cache. According to it, we have identified 1 $Friendly$, 2 $Fitting$ and 1 $Thrashing$ application. In the first step we estimate the partition sizes S_i ($S_{Friendly} = 8$, $S_{Fit.1} = 4$, $S_{Fit.2} = 3$ and $S_{Thrashing} = 1$) and we order them giving the highest priority to the $Friendly$ ones and lowest to $Thrashing$. The two $Fitting$ get their initial priority based on their S_i sizes. Overall, the highest priority application is inserted at MRU and the rest of

Table 7.1: Full-system detailed simulation parameters assumed for the evaluation of MCFQ.

Memory Subsystem	**L1 D + I Cache**	**L2 Cache**	**Main Memory**	**Memory Controller**	**Prefetcher**
	2-way, 64 KB, 3 cycles access time, 64 Bytes cache block size	16-way, 4 MB, 12 cycles bank access, 64 Bytes cache block size	8 GB, 16 GB/s, DDR3-1066-6-6-6, 16 Requests per Core	2 Controllers, 2 Ranks per Controller, 32 Read/Write Entries	H/W stride n, 8 streams / core, prefetching in L2
Core Parameters	**Clock Frequency**	**Pipeline**	**Reorder Buffer /Scheduler**	**Branch Predictor**	
	4 GHz	30 stages / 4-wide fetch / decode	128/64 Entries	Direct YAGS / indirect 256 entries	

IPs are based on the partition sizes from higher to lower priority. Since we have 2 *Fitting* applications, in *Step* 2 we estimate their *Sensitivity Factors* and order them from higher to lower. Assuming that *Fitting* 2 got a higher value we swap the two applications with the proper change of their IPs. Finally, in *Step* 3, we use the *Partitions Scaling Scheme* to scale their partitions based on their average cache-way occupancies measured on last-level cache during the last epoch (first couple of epochs we do not perform scaling since we are still warming up the profilers). Figure shows the finally selected IPs assuming we have actually scaled the partitions based on profiled data. *Step* 3 shows the final quasi-partitions assumed for every applications. For example, *Fitting* 2 inserts new lines in the 12^{th} from LRU cache-way and shares ways 12^{th} to 8^{th} only with the higher priority *Friendly* and ways 8^{th} to 2^{nd} with both the *Friendly* and *Fitting* 1. Finally, the first LRU way, is shared by all of the applications including the *Thrashing*.

7.4 Evaluation

To evaluate the scheme, we simulated a 4 and 8 cores CMP system using Simics functional model [48] extended with GEMS tool set [49]. GEMS provides an out-of-order processor model along with a detailed memory subsystem that includes an interconnection network and a memory controller. The default memory controller was augmented to simulate a *First-Ready, First-Come-First-Served* (FR_FCFS) [66] controller configured to drive a DDR3-1066 DRAM memory. Finally, we implemented a size *N* stride prefetcher that supports 8 streams per core. Table 8.2 summarizes the full-system simulation parameters.

We used multi-programmed workloads using mixes of SPEC CPU2006 suite [16] that are shown in Table 7.2 for 4 and 8-cores CMP. The workload mixes combine benchmarks with different cache behaviors (*Friendly*, *Fitting* and *Thrashing*) and levels of MLP. To estimate a representative average behavior, for each one of the experiments we simulated 8 segments of 100M instructions, evenly selected along the whole execution of the shortest benchmark of each experiment set. To do so, we fast-forward each benchmark to the beginning of each segment; use the next 100M instructions to warm up the caches and memory controller structures; and then use the following 100M instructions for the evaluation. The performance of each experiment is estimated based on the average behavior over the 8 segments.

The MCFQ behavior is compared against three previously proposed schemes: a) a cache partitioning scheme based on isolated partitions: *Utility-based Cache Partitioning* (UCP) [63], b) a cache pseudo-partitioning scheme: *Promotion Insertion Pseudo-Partitioning* (PIPP) [86], and c) a dynamic cache line insertion policy: *Thread-aware Dynamic Insertion Policy* (TADIP) [26]. More information about the techniques can be found in Section 2.2. We implemented all these schemes by modifying GEMS' memory model (Ruby). For each experiment we present the throughput and the fairness estimated

Table 7.2: Multi-programed benchmark sets from SPEC CPU2006 [16] suite for 4 and 8 cores.

4 Cores		8 Cores	
Benchmark Group	**Benchmarks**	**Benchmark Group**	**Benchmarks**
Mix 1 - All Friendly	soplex, bzip2, h264ref, perlbench	Mix 1- All Friendly	soplex, omnetpp, perlbench, calculix, gromacs, dealII, calculix, gromacs
Mix 2 - All Fitting	xalancbmk, wrf, tonto, gamess		
Mix 3 - All Thrashing	leslie3d, sjeng, bwaves, zeusmp	Mix 2 - All Fitting	xalancbmk, gobmk, wrf, gobmk, hmmer, astar, gamess, hmmer
Mix 4 - 3 Fr.:1 Fit.	omnetpp, bzip2, calculix, astar		
Mix 5 - 2 Fr.:2 Fit.	bzip2, mcf, gobmk, gamess	Mix 3 - 4 Fr.:2 Fit.:2 Thr.	omnetpp, bzip2, gobmk, gromacs, povray, h264ref, lbm, libquantum
Mix 6 - 1 Fr.:3 Fit.	omnetpp, xalancbmk, gamess, wrf		
Mix 7 - 3 Fr./Fit.:1 Thr.	mcf, perlbench, hmmer, bwaves	Mix 4 - 2 Fr.:4 Fit.:2 Thr.	mcf, gobmk, gromacs, hmmer, gamess, tonto, libquantum, milc
Mix 8 - 2 Fr./Fit.:2 Thr.	xalancbmk, dealII, milc, zeusmp		
Mix 9 - 2 Fr.:1 Fit.:1 Thr.	mcf, bzip2, astar, leslie3d	Mix 5 - 2 Fr.:2 Fit.:4 Thr.	omnetpp, soplex, gobmk, gamess, libquantum, milc, zeusmp, milc
Mix 10 - 1 Fr.:2 Fit.:1 Thr.	mcf, gobmk, gamess, libquantum		

as:

$$Throughput = \sum_{i=0}^{\#Cores} IPC_i \tag{7.8}$$

$$Fairness = \frac{N}{\sum_{i=0}^{\#Cores} \frac{IPC_{i,alone}}{IPC_{i,shared}}} \tag{7.9}$$

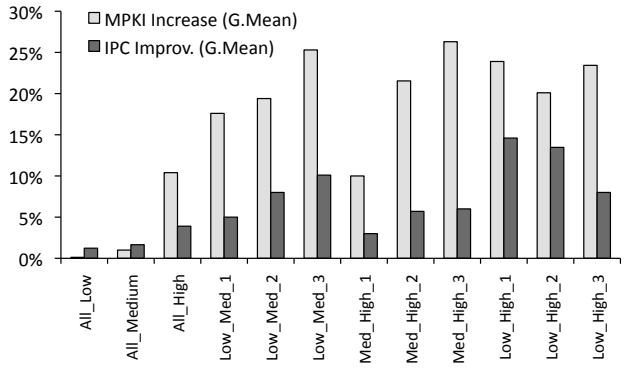
where $IPC_{i,alone}$ is the IPC of the i-th application when it was executed stand-alone with exclusive ownership of all the resources. The fairness metric expressed as the harmonic mean of weighted speedup of each applications was previously proposed in [47] after that has been used by the majority of papers evaluating execution fairness.

7.4.1 MLP-aware Capacity Assignment

We first evaluate the effectiveness of the MLP-aware cache capacity assignment algorithm (Section 7.3.2.1). To do so, we extended UCP isolated partitioning scheme using the$Utility_Rate$ function and compared it against the original UCP implementation. Fig. 7.3 provides a comparison of the approach against simple UCP for 12 specific experiments (Table 7.3(b)) executed on a 4-core CMP system. To evaluate the scheme, we need combinations of benchmarks with low, medium and high levels of MLP. To do so, we segmented the benchmarks of SPEC CPU2006 in three categories based on their measured MLP. *Low* MLP threshold was set to 2; *Medium* selected between 2 and 4; and higher than MLP of 4 was characterized as *High*. The experiments were selected to cover most of the interesting combinations on a 4-core system. Overall, we included cases for which a) the MLP of all benchmarks is approximately the same (first 3 experiments), b) there is a small difference in MLP between the benchmarks in the set (next 6 experiments), and c) there is a combination of benchmarks with significant big variation of MLP (last 3 experiments).

7.4.2 MCFQ – Performance evaluation on a 4 & 8 cores CMP

Fig. 7.3 includes the IPC improvement and the last-level cache's *Misses Per Thousand Instructions* (MPKI) degradation over simple UCP. Since the target was to improve the final performance and not to reduce the absolute number of misses, in all cases we actually happened to increase the number of misses but, in parallel, improved performance. The selection of benchmarks targeted the extreme cases to show the potentials of the scheme. In normal use, the MPKI is not expected to be so drastically increased. Despite that, the IPC is shown to improve up to 15% with an average (Geometric Mean) improvement close to 8% for our experiments. The benchmarks of the first three cases have comparable MLP and therefore, each application's cache misses are



(a) IPC and MPKI comparisons over UCP scheme.

Name	Description	Benchmark Set
All_Low	4 Low	gcc, perlbench, h264, astar
All_Medium	4 Medium	dealII, xalancbmk, gobmk, hmmer
All_High	4 High	bzip2, soplex, bwaves, mcf
Low_Medium_1	1 Low – 3 Medium	namd , dealII, sjeng, calculix
Low_Medium_2	2 Low – 2 Medium	sphinx3, GemsFDTD, tonto, povray
Low_Medium_3	3 Low – 1 Medium	sphinx3, h264ref, cactusADM, libquantum
Medium_High_1	1 Medium – 3 High	gobmk, soplex, mcf, leslied3d
Medium_High_2	2 Medium – 2 High	tonto, calculix, bwaves, omnetpp
Medium_High_3	3 Medium – 1 High	dealII, gobmk, hmmer, mcf
Low_Hgh_1	1 Low – 3 High	h264ref, bzip2, omnetpp, soplex
Low_High_2	2 Low – 2 High	astar, gcc, leslie3d, bzip2
Low_High_3	3 Low – 1 High	h264ref, astar, perlbench, omnetpp

(b) Experiments to evaluate MLP-assignment description.

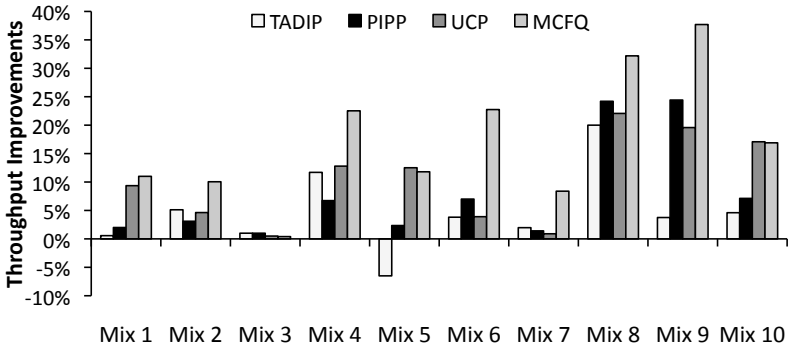
Figure 7.3: Evaluation of MLP-aware assignment of cache capacity assuming an isolated cache partitioning scheme like UCP [63] on a 4core CMP system.

equally important for performance. As a result, the performance of the proposed scheme is very close to UCP's. In the cases of combining benchmarks with a small difference in MLP, that is *Low-Medium-1* to *Medium-High-3*, there is a significant gain in IPC close to 8% for *Low-Medium* and 6% for *Medium-High* categories. For these cases, the results show that the IPC is improved more when a small number of benchmarks with high MLP executes in the set; cases *Low-Medium-3* and *Medium-High-3*. That is a strong proof that, the high MLP benchmark in the case of simple UCP, was granted a bigger partition than

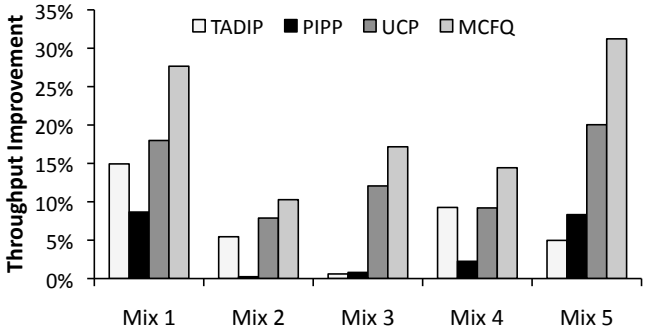
what it should have been assigned. The main reason of such assignment is the use of the absolute number of misses to estimate its *Marginal Utility*. Such bigger partition is not actually contributing to performance, restricting the rest of the lower MLP benchmarks by occupying useful cache capacity. The proposed approach can recognize the impact of each application to final performance and effectively assign more capacity to the lower MLP benchmarks. Finally, the biggest performance improvements are found in the last three categories where there is a significant difference between the MLP. The best IPC improvement took place for the case of 3 *High* MLP benchmarks. The proposed scheme can effectively recognize the importance of the lower MLP benchmark, *h264ref* in that case, and avoid allocating most of cache capacity to the higher MLP benchmarks that generate the majority of cache misses. The last two experiments followed with slightly smaller improvements that are proportional to the number of benchmarks with a big difference in MLP.

Fig. 7.4(a) includes the throughput improvements TADIP, PIPP, UCP and MCFQ schemes achieved over simple LRU with no advanced cache management scheme, for the benchmark sets listed in Table 7.2 in the case of a 4-core system. To get the baseline behavior for each category, the first three mixes include applications from the same cache-behavior category; the next three mixes contain combinations of only $Friendly$ and $Fitting$ and finally the last four of them target interesting combinations of all behaviors.

Overall, MCFQ demonstrates significant improvements over the next best scheme of every category. The only cases where MCFQ is comparable to other schemes are Mix 3, Mix 5 and Mix 10. In Mix 3, all benchmarks are $Thrashing$ and therefore all schemes except UCP have chosen to insert new lines in MRU position, so performance is almost the same with LRU. UCP had evenly allocated the cache to all four applications but since the benchmarks are $Thrashing$ the cache, it could not get any real benefit out of it. On the other hand, Mix 5 and 10, include 2 $Fitting$ benchmarks and all schemes



(a) 4 cores Throughput.



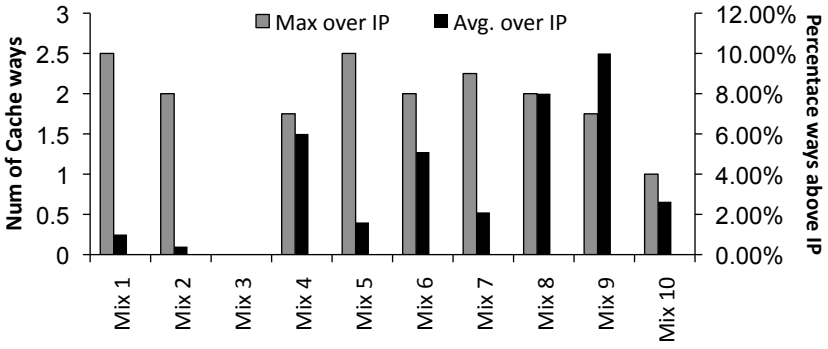
(b) 8 cores Throughput.

Figure 7.4: Comparison of throughput improvements of MCFQ, TADIP, PIPP and UCP schemes for 4 and 8 cores over simple, no cache management LRU scheme.

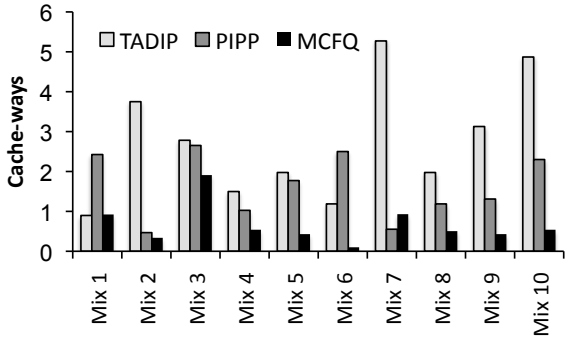
expect UCP, did worse than MCFQ because they could not guarantee a minimum number of ways for the *Fitting* cores to avoid them entering the *Thrashing* behavior. UCP and MCFQ managed to do equally well by minimizing the interference and providing enough capacity to the *Fitting* applications. Overall, UCP did well in most cases that include a *Fitting* application, even better than PIPP, which was initially unexpected. Looking carefully at the statistics, Fig. 7.5(a) shows that for these cases, PIPP effectively used the excess of space in the MRU positions only in *Mixes* 6, 8 and 9. For the rest of the cases, the space was practically unused, wasting space that UCP effectively used to improve performance. Furthermore, when multiple *Fitting* applications had almost the

same partition size, PIPP's insertion policy put them on the same IP; introducing severe contention and therefore forcing them to work in their *Thrashing* area. This is clear from the fact that PIPP is better than UCP in *Mix* 6, 8 and 9 where only one fitting application exists. Despite that, MCFQ was still significantly better than PIPP in the same categories. Fig. 7.5(b) shows how far away from ideal is the average cache-way occupancy for each technique. From the figure, MCFQ gains were mainly due to the priority scheme and scaling of partitions to ensure correct average occupancy of cache ways. Finally, TADIP had an intermediate behavior by achieving comparable results to the second best in a small number of cases. Unfortunately, due to its policy to either insert a new line in the MRU or LRU position, it cannot properly handle *Fitting* applications; while in *Mix* 5 it is even worse than simple LRU by 7%. As expected, TADIP gets reasonable performance improvements only in cases with *Thrashing* applications. Overall, MCFQ for the specific 4-cores cases, managed an average improvement of 19%, 14%, 13% and 10% over LRU, TADIP, PIPP and UCP, respectively.

The 8-cores results in Fig 7.4(b) can potentially show how well each scheme can scale with the number of cores. Notice that since we have a 16-way last-level cache, each core in a 8-core CMP can get on average 2 ways. Therefore, these benchmark mixes put a lot of pressure on each scheme to keep the most important cache-lines for each application in the cache. As expected, UCP is the second best followed by TADIP. UCP can effectively choose the best isolated partitions size to improve performance while TADIP can handle the high demand rates by forcing the new lines from the applications that hurt performance the most to be allocated at the LRU positions. Both schemes though got gains by helping only a small number of threads, restricting the rest of applications to only one cache-way; hurting overall system fairness. PIPP showed the worst behavior since there are many applications with similar partition sizes that insert lines in the same IP. The statistics show that in most cases the applications were evicting each other's cache-lines, forcing them to



(a) PIPP's maximum and average number of cache-ways promoted higher than applications' IPs (Avg. over all cores).



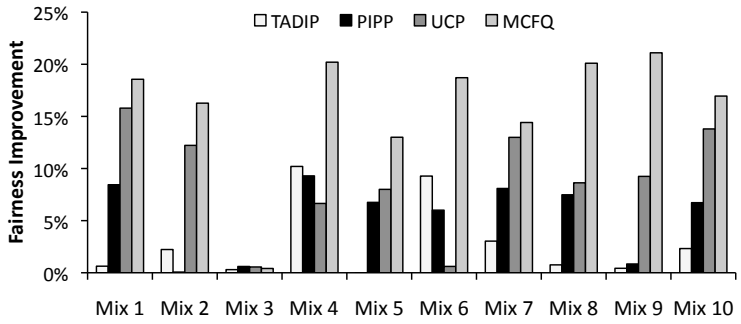
(b) Average absolute error from ideal cache-way occupancy (UCP is always 0).

Figure 7.5: Cache resource use statistics for the 4-core CMP runs.

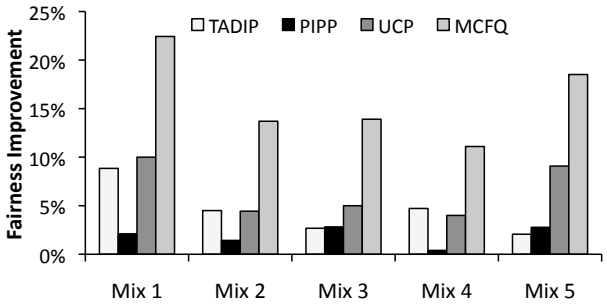
a thrashing execution style. Overall, MCFQ showed significant throughput improvements due to its ability to share the capacity efficiently while minimizing the interference. For the 8-cores cases, MCFQ achieved an average improvement of 20%, 13%, 17% and 8% over LRU, TADIP, PIPP, and UCP, respectively.

7.4.3 MCFQ – Fairness evaluation on a 4 & 8 cores CMP

Fig. 7.6 provides a comparison of all schemes using the harmonic mean of weighted speedup as a fairness metric. By comparing Fig. 7.4(a) and Fig 7.6(a) one can



(a) 4 cores Fairness.



(b) 8 cores Fairness.

Figure 7.6: Comparison of performance fairness improvements of MCFQ, TADIP, PIPP and UCP schemes for 4 and 8 cores over simple, no cache management LRU scheme.

see that the fairness of MCFQ is improved across the whole set of benchmarks even for the cases that MCFQ had comparable performance gains to other schemes (*Mix* 5 and 10). In addition, even for the cases that TADIP, PIPP and UCP provide a significant performance improvement over LRU, such improvement is coming from helping only a small number of threads and restrict the others. As a result, their fairness is significantly lower than MCFQ's.

TADIP and PIPP cannot efficiently target fairness because of their greedy philosophy to help the applications that can get the most performance gains and they have no real protection mechanisms to ensure fair execution. UCP on the other hand, could be

configured to provide fairness with proper allocation of capacity in its isolated partitions, but such approach would be wasteful and it has been shown in that past that will hurt performance [20]. Therefore, UCP can either help throughput or fairness. Such behavior can also be confirmed by looking into the 8-core fairness results of Fig. 7.6(b). TADIP and UCP managed to get their performance gains by helping only a subset of applications. For 8-cores, the worst overall behavior was achieved by PIPP because under high pressure, its insertion point policy fails to help the competing threads, hurting its performance fairness. In addition, PIPP's inherent difficulty to fairly scale can be observed by looking the severe reduction of fairness when we scale from 4 to 8 cores.

MCFQ, on the other hand, can target performance fairness by allocating capacity based on applications' performance sensitivity to cache space and misses. In addition, MCFQ's careful handling of *Fitting* applications seems to improve fairness for the cases where many Fitting applications coexist (*Mix* 2, 6, 8 and 10). MCFQ achieved its best fairness performance in the cases where many *Friendly* applications were competing for capacity. For these cases, the *Interference Sensitivity Factor* could effectively reduce interference across the same category of applications and the *Partitions Scaling Scheme* was able to fairly scale the capacities allocated to them. Overall, for the 4 core cases, MCFQ achieved an average improvement of 17%, 12%, 14% and 9% over LRU, TADIP, PIPP and UCP, respectively. Finally, for the 8 cores cases, the improvements were found to be 15%, 13%, 12% and 8% over LRU, TADIP, PIPP, and UCP, respectively.

7.5 Summary

This chapter presents MCFQ, a last-level cache quasi-partitioning scheme that effectively allocates capacity to resource competing applications, while minimizing destructive interference. MCFQ by utilizing applications' memory-level parallelism can predict applications' performance sensitivity to last-level cache misses and therefore,

target final system throughput improvements. MCFQ significantly reduces the effects of applications' shared cache interference by categorizing and assigning priorities according to applications' *cache friendliness* behavior (Friendly, Fitting, Thrashing) and their performance sensitivity on sharing cache capacity (*Interference Sensitivity factor*). Finally, to further improve throughput and overall fairness, MCFQ monitors the average occupancy of cache capacity per application and, using the proposed *Partition Scaling Scheme*, adjust their quasi-partitions sizes to bring the average cache-ways occupancy of each application closer to the ideal estimated ones. Overall, MCFQ demonstrates significant improvements over previously proposed representative schemes, achieving on a 4-core CMP improvements up to 18% in throughput (10% average) and 23% fairness (9% average) over the next best scheme.

Chapter 8

A Criticality-based DRAM Memory Controller Scheme Based on a hybrid open/close Page-mode Policy

Contemporary DRAM systems have maintained impressive scaling by managing a careful balance between performance, power, and storage density. In achieving these goals, a significant sacrifice has been made in DRAM's operational complexity. DRAM's efficient use is further complicated in many-core systems where the memory interface has to be shared among multiple cores/threads competing for memory bandwidth.

The memory controller is a fundamental component of the memory-subsystem that is responsible of coordinating the operation of the main memory with the stream of memory requests from the last-level cache. To achieve an efficient use of DRAM, it is crucial that the memory controller policy is able to identify and issue with high higher priority to the memory the requests that are more critical to performance and/or fairness. Such operation therefore has to be synchronized with the DRAM "page-mode" policy and the memory requests sources of a typical CMP, that is the last-level cache and the prefetch unit.

The use of the "Page-mode" feature of DRAM devices can mitigate many DRAM constraints. Current open-page policies attempt to garner the highest level of page hits. In an effort to achieve this, such greedy schemes map sequential address sequences to a single DRAM resource. This non-uniform resource usage pattern introduces high levels of conflict when multiple workloads in a many-core system map to the same set of resources. This dissertation recognizes that page-mode access gains are realized with only a small

number accesses per activation and therefore, assumes a scheme that targets "just enough" page-mode accesses to garner page-mode benefits, avoiding system unfairness.

This chapter presents a criticality-based memory request priority scheme that ranks demand read and prefetch operations based on their latency sensitivity of each operation. It uses a fair memory hashing scheme to control the maximum number of page mode hits, and direct the memory scheduler with processor generated prefetch meta-data. An evaluation of the scheme across a range of memory utilization workload mixes demonstrated gains in throughput of 6-7% over the best prior proposals for medium and high memory utilization levels, in conjunction with improved fairness.

8.1 Introduction

Since its invention, the DRAM memory subsystem has proven to be one of the most important system components. The requirement of a properly designed memory subsystem is further amplified in the case of chip-multiprocessors where memory is shared among multiple, concurrently executing threads. An improperly managed memory can lead to significant degradation of both individual threads performance as well as overall system throughput. In the many-core era, DRAM requests scheduling becomes critically important. As the memory interface is shared among the growing number of cores, providing both sustained system throughput and thread execution fairness is equally critical. To do so, the memory controller must be able to make an intelligent selection of requests to send to memory at any given point. This selection must carefully balance thread execution speed and overall throughput, functioning well across a broad range of memory utilization levels.

To provide an efficient balance between memory density, request latency and energy consumption, DRAM designers have adopted a complex architecture that imposes

a number of structural and timing limitations. The memory controller policy is a fundamental component of the memory-subsystem that is responsible of coordinating the operation of the main memory with the last-level cache and the underline processors. Effectively, a memory controller consists of a buffer that buffers the memory requests while they wait to be serviced and a scheduler that selects the next request to be sent to the memory. The scheduler has to consider both the state of the DRAM banks and the available busses along with the state of each request before choosing one to be served. *Ready* requests in the queue have to avoid any DRAM resource conflicts (banks, ranks, buses) and do not violate any DRAM timing constrains. Therefore, to achieve an efficient use of DRAM, it is crucial that the memory controller is able to identify and issue with high higher priority to the memory the requests that are more critical to performance and/or fairness while respecting all the complex DRAM specification restrictions.

One of the most important components of the DRAM system is the *Row Buffer*. The row buffer serves two primary purposes. As each DRAM bit is maintained as an electrical charge stored in a capacitor, cell reads are destructive. For each DRAM access, the entire data word is captured in the row buffer such that the data can be restored for future accesses. In addition, the row buffer functions as the interface between the very wide access width of the DRAM array (1000kBytes) and the few IO pins connected to the processor (4-16 bits) [24]. The row buffer can service multiple data transfers from much wider DRAM cell access or "activate". These row buffer or "page mode" accesses can effectively amortize the high cost of the DRAM cell reads across multiple data transfers, improving system performance and reducing DRAM power consumption.

One primary parameter in controlling the use of row-buffer is the page-mode policy which tend to be grouped into two classes. Leaving a row buffer open after every access (*Open-page* policy), enables more efficient access to the open row, at the expense of increased access delay to other rows in the same DRAM array. *Closed-page*

policies avoid the complexities of the row buffer by issuing a single access for each row activate [70]. This class of policies provide a consistent fair latency at the expense of potential page mode gains. *Open-page* policies attempt to gather multiple requests into each row buffer access by speculatively delaying the precharge in an effort to execute additional row reads. This enables latency, scheduling, and power improvements possible with page mode accesses. However, as these policies are applied to the numerous request streams of a many-core system, priority is given to accesses to already opened pages, introducing memory requests priority inversion and potential thread fairness/starvation problems [39, 40, 53, 56].

In conjunction with the page policy, the mapping of the memory data address to the DRAM device address (row, column, bank) must be properly structured to maximize efficiency. In a closed-page policy, sequential cache block accesses cycle through the multiple DRAM arrays or "banks" contained in each DRAM chip first, reducing costly back to back requests to the same bank. In an open-page policy, sequential cache blocks are mapped to the same DRAM row in an effort to obtain the greatest amount of row buffer hits [39, 45, 56, 66]. While the open page mapping policy works well for single core systems, the system can degrade when multiple workloads conflict [53]. In this case, the mapping of many sequential accesses to a row becomes a source of significant performance degradation.

This chapter presents a intuitive criticality-based memory request priority scheme that ranks demand read and prefetch operations based on their latency sensitivity of each operation. Since an efficient implementation of a memory controller is tightly connected with the DRAM page-mode policy and memory requests inherent latency criticality, the presented scheme takes both into account and suggests a coordinated scheme. Its operation is based on two key observations: a) Page mode gains can be realized with a relatively small number of page accesses for each activation that can be achieved through a DRAM

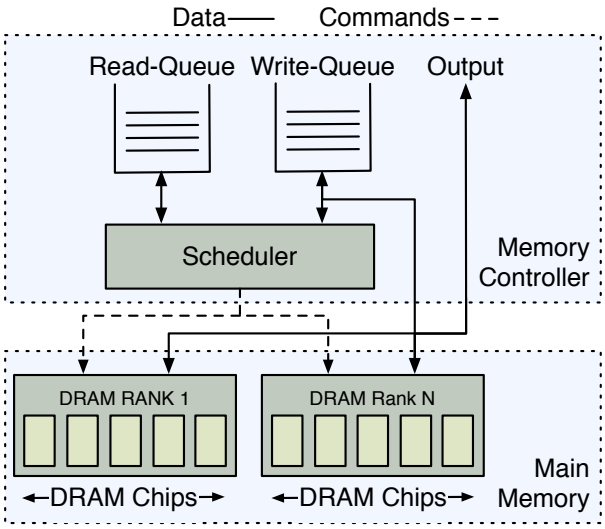


Figure 8.1: Typical memory controller organization driving DDRx main memory modules.

address mapping scheme that targets fairness, and b) page mode hits exploit spatial reference locality, of which the majority can be captured in modern prefetch engines. Therefore, the prefetch engine can be used to explicitly direct page-mode operations in the memory scheduler. Such system can essentially guarantee that the target page hit rate will be met, irrespective of conflicts between competing execution threads. Finally, by providing fairness through the DRAM mapping policy allows the scheduler to focus its priority policy directly on the memory request criticality, which is important in achieving high system throughput and fairness.

8.2 Motivation

A typical memory controller organization is shown in Figure 8.1. This dissertation proposes optimizations to improve the access latency and increase the sustained efficiency of the IO interface connecting main memory with the processors. This is accomplished

through policy improvements to the memory scheduler and the way such memory operations are mapped to the DRAM devices. More information regarding main memory and memory controller operation can be found in Section 2.1.3.

DRAM devices operate on large, 1KB pages (also referred to as *rows*). Each DRAM array access causes all 1KB of a page to be read into an internal array called *Row Buffer*, followed by a "*column*" access to the requested sub-block of data. Since the read latency and power overhead of the DRAM cell array access have already been paid, accessing multiple columns of that page decreases both the latency and power of subsequent accesses. Successive accesses are said to be performed in *page mode* and the memory requests that are serviced by an already opened page loaded in the row buffer are characterized as *page hits*. Due to the reductions in both latency and energy consumption possible with page mode, techniques in the memory controller policy that aggressively target page mode operations are often used. There are downsides however, which must be addressed. Leaving a specific page *open* produces a higher access latency to other rows in the same bank (*page conflict*) . In addition, certain scheduling algorithms such as the widely used *First-Ready, First-Come-First-Served* (FR-FCFS) [66] give higher priority to page hit operations in my controller queue, which can result in unfairness for non page hit operations. Therefore, although row buffer hits are useful, they must be used in moderation.

8.2.1 Balanced Designs and Importance of Cache Capacity and Memory Bandwidth

While there is a number of previously proposed memory scheduling algorithms that focused on improving either fairness, throughput or combination of both, they all assume DRAM-centric approaches that base their operation on knowledge collected from a significant number of requests being queued in the memory controller. An efficient

design requires each of the structures in the system to be appropriately balanced. If certain components are over-sized, these resources are wasted while undersizing structures create bottlenecks. As analyzed in "Scaling the bandwidth wall" [67], the balance between number of cores, cache capacity, and memory bandwidth must scale appropriately into future generations. In practice, a system will service workloads with a wide range of cache capacity requirements and memory bandwidth pressure. Therefore, a well designed system will saturate the memory interface on a reasonable subset of workloads. However, this leaves an important fraction of workloads where the bandwidth usage is significant, yet not saturated.

This insight has important implications into systems design. Specifically, the presented scheme found that the fullness of the read input queues of the memory controller is relatively low for many workload combinations. The only cases where there is a significant queue depth is when several high bandwidth workloads execute together, saturating the memory interface. Due to the low fullness of the memory structures in these cases, policies which employ reordering of requests in the memory controller as the primary control point are inherently ineffective for workloads where the memory interface is below saturation. This is a key motivator for this work.

8.2.2 Benefits of DRAM Page-mode operation

The potential benefits of memory operations that are taking advantage of the page-mode are the following:

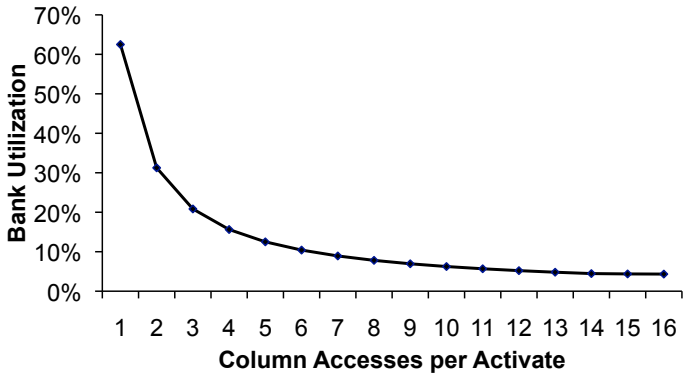
1. **Latency Effects:** Overall, increases in latency are caused by two mechanisms in the DRAM. Firstly, if a *row* is left open in an effort to service page hits, to service a request to another page incurs a delay of 12.5ns to close the current page followed by the latency to open and access the new page. Secondly, DRAM devices specify a minimum delay between back-to-back activations of two different rows within the same

bank. This delay, known as *tRC* parameter, has remained approximately 50ns across the most recent DDRx DRAM devices. In a system that attempts to exploit page mode accesses, the overall effect on loaded memory latency and program execution speed due to the combination of these properties can significantly increase the observable latency.

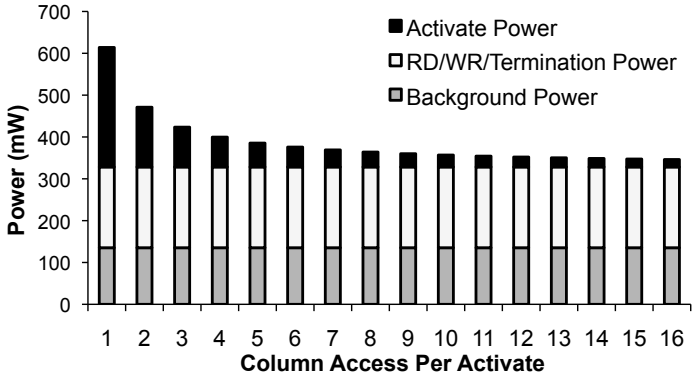
2. **Bank Utilization:** The utilization of the DRAM banks can be a critical parameter in achieving high scheduling efficiency. If bank utilization is high, the probability that a new request will conflict with a busy bank is greater. Increasing the data transferred with each DRAM activate, through page mode, amortizes the expensive DRAM bank access, reducing utilization. Figure 8.2(a) shows the bank utilization of a DDR3 1333 MHz system, with two devices (*ranks*) sharing a data bus at 60% bus utilization. A closed-page policy, with one access per activate would produce an unreasonably high bank utilization of 62%. However, the utilization drops off quickly as the accesses per activate increases. For example four accesses per activate reduces the bank utilization to 16%, greatly reducing the probability that a new request will be delayed behind a busy bank.

3. **Power Reduction:** Page mode accesses reduce DRAM power consumption by amortizing the activation power associated with reading the DRAM cells data and storing them into the row buffer. Figure 8.2(b) shows the DRAM power consumption of a 2GBit DDR3 1333MHz DRAM as the number of row accesses increases. Since page mode only reduces the page activation power component, DRAM power quickly becomes dominated by the data transfer and background (not proportional to bandwidth) power components.

4. **DRAM Design Complexities:** More subtle DRAM timing rules can have significant effects of DRAM utilization, especially as the data transfer clock rates increase in every DRAM generation. Many DRAM parameters do not scale with frequency increases



(a) Bank utilization for a 2-Rank 1333 Mhz system at 60% data bus utilization.



(b) DRAM power for each 2GBit DDR3 1333 Mhz at 40% read, 20% write utilization [51].

Figure 8.2: Analysis of power and bus utilization improvements as a function of the number of DRAM bank accesses per activation.

due to either constant circuit delays and/or available device power and other physical restrictions. One example is the TFAW parameter. TFAW specifies the maximum number of activations in a rolling time window in order to limit peak instantaneous current delivery to the device. In a 1333MHz DDR3, the TFAW parameter specifies a maximum of four activations every 30ns. A transfer of a 64 byte cache block requires 3ns, thus for a single transfer per activation TFAW limits peak utilization to 80% ($6ns * 4/30ns$). However, with only two accesses per activation, TFAW has no effect ($12ns * 4/30ns > 1$). The same trend is observed across several other DRAM parameters, where a single access per activation results in efficiency degradation, while a small number of accesses alleviates the restriction.

In summary, a relatively small number of accesses to a page is found to be very effective in taking advantage of DRAM page mode for both scheduling and power efficiency. For example, at four row accesses per activation, power and bank utilization are 80% of their ideal values. Therefore, the presented scheme uses a hybrid page-policy scheme that allows only four accesses per activation before closing a page. Unfortunately, latency effects are more complex, as scheduling policies to increase page hits also increase bank conflicts, making raw latency reductions difficult to achieve. The following section provides a short description.

8.2.3 Bank and Row Buffer Locality Interplay With Address Mapping

Many high performance processors (eight in current leading edge designs) are backed by large last-level caches containing up to 32 MB of capacity [30]. A typical memory hierarchy that includes the DRAM row buffer is shown in Figure 8.3. As a large last-level cache filters requests to memory, row buffers inherently exploit only spacial locality. Temporal locality within the program results in hits to the much larger last-level cache. Access patterns with high levels of spatial locality, that miss in the large last level

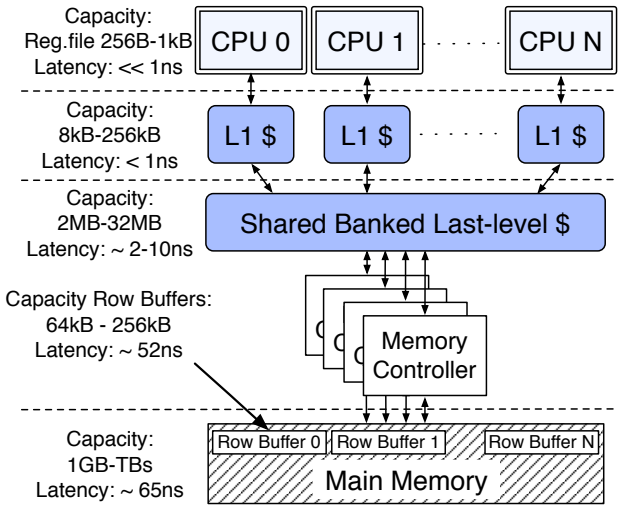


Figure 8.3: Typical capacities and access latencies of every level in memory hierarchy.

cache, are often very predictable. In general, speculative execution and more specifically modern prefetch algorithms can be exploited to generate memory requests with spatial locality in dense access sequences. Consequently, the relatively small latency benefit of page mode is hidden.

Commonly used open-page address mapping schemes put all column bits directly above the cache block to capture the greatest possible number of page hits [39, 45, 56, 66]. As identified by Moscibroda *et al.* [53], this hashing can produce interference between the applications sharing the same DRAM devices, resulting in significant performance loss. The primary problem identified in that work is due to the FR-FCFS [66] policy where page hits have a higher priority than requests with lower page affinity. Beyond fairness, schemes that map long sequential address sequences to the same row, suffer from low bank-level parallelism (BLP). If many workloads with low BLP share a memory controller, it becomes inherently more difficult to interleave the requests, as requests from two workloads mapping to the same DRAM bank will either produce a large number of

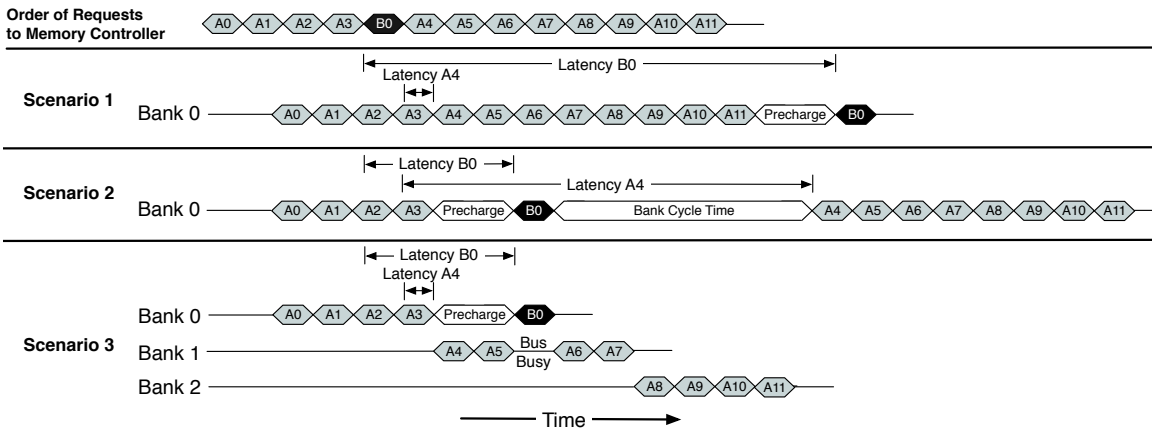


Figure 8.4: DRAM Row buffer policy examples.

bank conflicts, or one of them has to stall, waiting for all of the other workload's request to complete, significantly increasing it's access latency.

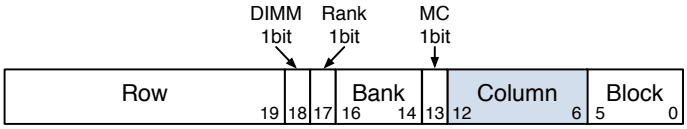
Figure 8.4 illustrates an example where workload A generates a long sequential access sequence, while workload B issues a single operation mapping to the same DRAM. With a standard open-page policy mapping, both requests map to the same DRAM bank. With this mapping, there are two scheduling options, shown in Scenarios 1 and 2. The system can give workload A higher priority until all page hits are completed, significantly increasing the latency of the workload B request (Scenario 1, Figure 8.4). Conversely, workload A can be interrupted, resulting in very inefficient activate to activate commands conflict for request A4 (Scenario 2, Figure 8.4), mainly due to the time to load the new page in the row buffer and the unavoidable *tRC* timing requirement between back-to-back activations of a page in a bank. Neither of theses solutions optimize fairness and throughput. In the proposal we adapt the memory hash to convert workloads with high row buffer locality (RBL), into workloads with high bank-level parallelism. This is shown in Scenario 3 of Figure 8.4, where sequential memory accesses are executed as reading four

cache blocks from each row buffer, followed by switching to the next memory bank. With this mapping, operation B can be serviced without degrading the traffic to workload A.

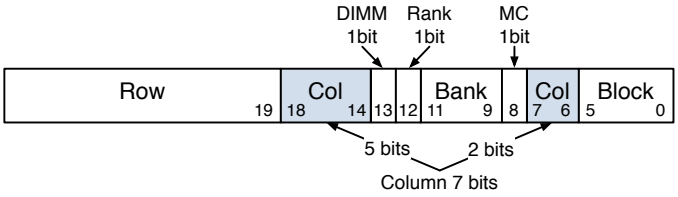
8.3 A Criticality-based DRAM Memory Controller Scheme Based on a hybrid open/close Page-mode Policy

As analyzed in the motivation section, we base the proposed scheme on the observation that most of the page-mode gains can be realized with a relatively small number of page accesses for every page activation. The presented scheme defines a target number of page hits that enable a careful balance between the benefits (increased performance and decreased power), and the detractors (resource conflicts and starvation) of page-mode accesses. To alleviate the need to address the row-buffer starvation directly in the memory request priority scheme, an address mapping scheme that directly targets fairness is provided. Doing that, the scheduler can focus its priority policy on memory request criticality, which is important in achieving high system throughput and fairness. As described in Section 8.2.3, most of the memory operations with "page-mode" opportunities are the results of memory accesses generated through prefetch operations. Therefore, the prefetch engine is used to provide request meta-data information which directs the scheme's request priority scheme and the page-mode accesses.

In the remaining of this section, the fair address mapping scheme is described that enables bank-level parallelism with the necessary amount of row-buffer locality. This is followed by the prefetch hardware engine, as this component provides prefetch request priorities and prefetch-directed page mode operation. Finally, the scheduling scheme is described for assigning priorities and issuing the memory request to the main memory.



(a) Typical Mapping.



(b) Proposed Mapping.

Figure 8.5: System address mapping schemes to DRAM addresses - The example system in figure has 2 memory controllers (MC), 2 ranks per DIMM, 2 DIMMs per channel, 8 banks per rank and 64B cache-line block size.

8.3.1 DRAM Address Mapping Scheme

The differences between a typical mapping and the one we use in this proposal are summarized in Figure 8.5. The basic difference is that the *Row Column* access bits that are used to select the row buffer columns are split in two places. The first 2 LSB bits (Least Significant Bits) are located right after the *Block* bits to allow the sequential access of up to 4 consequent cache lines in the same page. The rest of the MSB (most significant bits) column bits (five bits in this case-study since we assume 128 overall cache lines stored in every row buffer) are located just before the *Row* bits. Not shown in the figure for clarity, higher order address bits are XOR-ed with the bank bits shown in the figure to produce the actual bank selection bits. This reduces row buffer conflicts as described by Zhang *et al.* [87]. The above combination of bits selection allows workloads, especially streaming, to distribute their accesses to multiple DRAM banks; improving bank-level parallelism and avoiding over-utilization of a small number of banks that leads to thread starvation

and priority inversion in multi-core environments.

8.3.2 Data Prefetch Engine

To harvest the predictable page-mode opportunities, we need to utilize an accurate prefetch engine. The engine targets spatial locality prediction and is able to predict repeatable address strides. To do so, each core includes a hardware prefetcher that is able to detect simple access streams of stride 1, described by Lee *et al.* [42], along with "stride n" streams based on the stride prefetcher described by Doweck [10].

To keep a low bandwidth overhead and throttle prefetch aggressiveness, the prefetcher uses a prefetch depth distance predictor to decide how far from the currently accessed memory address each access stream should be prefetched. To avoid prematurely fetched data, the prefetcher "ramps up" gradually to the full depth only when there is confidence that an access stream is a useful, long access stream. To dynamically decide on the maximum depth of each access stream we utilize a structure based on the "Adaptive Stream Detection" (ASD) prefetcher from Hur *et al.* [22]. More specifically, we used the "Stream Length Histograms" (SHL) from ASD to decide on the depth of each access stream. ASD keeps a histogram of the number of read commands that was found to be part of a stream of length m. Depending on the detected stream length of the current read request, the prefetcher checks the SLH and determines whether to prefetch the next cache line by comparing the likelihood that a read request will be the last element of a stream against the likelihood that it will be part of a longer stream. As the length of a stream increases, ASD adjusts the maximum depth that we should prefetch along with the amount of time that we are willing to wait for the next element of a stream to appear. If this maximum waiting time is exceeded, the stream is deleted from the prefetcher. Finally, to avoid polluting the caches with unused prefetches, all the prefetched requests are stored in the LRU position of the last-level cache until used by executing instructions.

The prefetch engine is based around detection of predictable spatial address locality. Although there are more complex prefetch mechanisms available for predicting more irregular memory access patterns that can benefit system performance, the prefetcher matches the need for coordinating sequential accesses to the row buffer. More sophisticated prefetching schemes typically cover address ranges beyond the 8kByte DRAM page regions, making them unnecessary for the scheme.

8.3.2.1 Multi-line Prefetch Requests

Although the proposed prefetcher makes decisions on the granularity of a single cache-line, the scheme utilizes multi-line prefetch operations. A multi-line prefetch operation consist of a single request sent to the memory controller, to indicate a specific sequence of cache-lines to be read from a memory page. This policy was introduced in the IBM POWER6 design [42]. Multi-line operations reduce the command bandwidth and queue resource usage. Specifically for the proposed scheme, multi-line operations can consolidate the accesses to a DRAM in a controlled burst. That enables the issue of a single request to the memory controller queue; processing of multi-line requests that are directed to the same DRAM page together as a single request in the queue; and, in the end, close the page after completing all of the prefetches. Consequently, multi-line requests: a) improve bandwidth use, b) simplify the proposed priority scheme by executing back-to-back page-related requests, and c) improve the controller efficiency for closing the DRAM pages.

8.3.2.2 Multi-line prefetches combined with address mapping scheme

Since the mapping scheme targets a specific number of accesses to the same page, we direct the described multi-line prefetch requests to match the mapping. When the prefetcher identifies a prefetchable stream, it packs in the same multi-line request only the

prefetch requests that will go to the same page. This makes the processing of the multi-line requests easy and guarantees that all the prefetch requests to the same page can be processed together in the memory controller.

8.3.3 Memory Request Queue Scheduling Scheme

Previous priority-based open-page scheduling proposals either exhibit unfairness [66], or use request priority as a fairness enforcement mechanism [39, 40, 56]. For example, the ATLAS scheme [39] assigns the same priority to all of the requests of a thread in the memory queue based on attained service, assuming all requests from a thread are equally important. The proposed work found that in out-of-order execution, the importance of each request can vary both between and within applications. These range from requests that are critical for the performance (*e.g.* demand-misses) to requests that can tolerate more latency, such as misses in applications exhibiting high levels of Memory-level Parallelism (MLP) and prefetch requests. As fairness is solved through the address mapping scheme, priority is directed with each memory request's instantaneous priority, based on both the current MLP and metrics available within the prefetch engine.

8.3.3.1 DRAM Read Requests Priority Calculation

In the proposed memory scheduling priority scheme we assign a different priority to every memory request based on its criticality to performance. We separate the requests in two categories: a) *Normal read requests*, and b) *Prefetches*. Based on each request's category and its criticality to performance, the memory controller assigns to them an initial priority. To improve fairness, we implemented a *time-based dynamic priority scheme*. The scheme assigns an initial priority to every request and as a request remains in the queue waiting to be serviced, its priority is gradually increased. We use a three bit value as a priority indicator for every request. At a time interval of 100ns, each request's priority is

Table 8.1: Memory read requests priority assignment scheme.

Normal Requests		**Prefetch Requests**	
MLP level	**Priority (3bits)**	**Distance from Head**	**Priority (3bits)**
MLP $<$ 2 (Low MLP)	7	$0 \leqslant$ Distance $<$ 4	4
$2 \leqslant$ MLP $<$ 4 (Medium MLP)	6	$4 \leqslant$ Distance $<$ 8	3
MLP $\geqslant$ 4 (High MLP)	5	$8 \leqslant$ Distance $<$ 12	2
		$12 \leqslant$ Distance $<$ 16	1
		Distance $\geqslant$ 16	0

increased by one. Intuitively, a prefetch request has low priority when the request is first received, but as time passes, the program execution will reach the prefetch, effectively making the request a high priority.

Read requests are assigned higher priority than prefetches since the latency of demand misses is highly correlated to performance. We used the Memory-level Parallelism (MLP) information of the core that issued each request to identify criticality. The MLP information is directly collected from each core's *Miss Status Holding Registers* (MSHR) [41]. The MSHR tracks of all the outstanding L1 misses being serviced by the lower levels of memory hierarchy (L2 and main memory). As a result, the number of entries in each core's MSHR which indicate an L2 miss represents the current MLP of the application. Low MLP means that there is a small number of L2 outstanding misses and therefore each one is very important for the execution progress of the application. Any delay on serving these misses results in significant performance degradation. As MLP increases, there are more outstanding misses available but, on the average case, most of them do not block the progress of speculative execution on an out-of-order core, making their latency less important for performance. The algorithm statically assigns priorities by classifying the possible levels of MLP in three categories. The levels along with their

assigned priority are shown in the left part of Table 8.1.

Prefetch requests are assigned a priority level lower than normal requests using prefetch meta-data information sent by the prefetch engine. Their priority is based on the distance in cache blocks from the actual consuming instructions to the prefetch request. Requests with a small distance have higher priority, since they are more likely to be used in the near future. As the distance from the head of the stream increases, the prefetch's latency is less critical for performance. The priorities based on this distance are shown in the right part of Table 8.1.

8.3.3.2 DRAM Page Closure (Precharge) Policy

In general, the proposed memory controller policy does not speculatively leave DRAM pages open. If a multi-line prefetch request is being processed, the page is closed with an auto-precharge sent with the read command (In DDRX the auto-precharge bit indicates to close the page after the data are accessed [24]). This saves the command bandwidth of an explicit precharge command. For read and single line prefetch operations, the page is left open based on the following principle. The *tRC* DRAM parameter specifies the minimum time between activations to a DRAM bank. The *tRC* is relativity long at 50ns compared to the precharge delay of 12.5ns. Therefore, closing a bank after a single access does not allow reactivation of the bank until the *tRC* delay expires. With this insight, we speculatively leave pages open for the *tRC* window, as this provides for a "free" open page interval.

8.3.3.3 Overall Memory Requests Scheduling Scheme

The rules in Priority Rules 1 summarize the *per-request* scheduling prioritization scheme that is used in the proposed scheme. The same set of rules are used by all of the memory controllers in the system. As explained in Section 8.2.3, the proposed address

mapping scheme guarantees memory resource fairness while preventing starvation and priority inversion. Therefore, there is no need for any communication/coordination among the multiple controllers.

The scheduling scheme is based on assigning priorities to each requests individually based on their criticality to performance. Therefore the first, most important rule, is to schedule requests with the highest priority first. The second rule, namely "Ready-Requests First", guarantees that between requests with the same priority, requests that are mapped to the same temporally opened page are scheduled first. To clarify, if the controller is servicing the multiple transfers from a multi-line prefetch request, it can be interrupted by a higher priority request (assuming the needed bank is beyond *tRC*). This guarantees that requests that are very critical for performance can be serviced with the smallest latency, enabling the controller to work well in a wide range memory bus utilization levels.

8.3.3.4 Handling Write Operations

The dynamic priority scheme only applies to read requests as they directly limit the completion of instructions. While the completion of write requests does not directly effect an application's execution progress, the memory bandwidth consumed by memory writes and their interference with read requests' latency are still important components for performance. To alleviate the pressure of write requests, we follow an approach similar to the *Virtual Write Queue* (VWQ) proposal [74] in the handling of write operations inside the memory controller. This enables write request to avoid reads when possible, and causes minimal intrusion when the VWQ becomes full.

Priority Rules 1 Request Scheduling Rules in Memory Controller Queue.

1. **Higher Priority Request First:** Requests with higher priority/criticality are issue first in the proposed per-request scheme
2. **Ready-Requests First:** Requests that belong to the same multiline-prefetch request that is currently being serviced in a open bank are prioritize over other requests that are not "ready" for scheduling yet (Row-conflict/closed requests).
3. **First-Come First-Served:** Older requests issued first.

8.4 Evaluation

To evaluate the scheme, we simulated an 8 core CMP system using the Simics functional model [48] extended with the GEMS toolset [49]. We used an aggressive out-of-order processor model from GEMS along with a detailed memory subsystem. In addition, we modified the GEMS memory subsystem by adding the hardware prefetching engine described in Section 8.3.2. Finally, the default memory controller was augmented to simulate a DDR3 1333MHz DRAM using the appropriate memory controller policy for each experiment. Table 8.2 summarizes the full-system simulation parameters used in the study.

For the evaluation we utilize a set of multi-programmed workload mixes from the SPEC cpu2006 suite [16]. We selected 27 randomly created, 8-core mixes spanning from low bus utilization levels to saturation. To accomplish this we summed the single core bandwidth requirements of a large number of randomly selected workloads. We then selected 27 sets by choosing the total bandwidth target to span for 15% to 300%. The sets are ordered from lower to higher bus utilization. In addition, the workloads were divided in *low*, *medium*, and *high* sets with nine workloads each. The bandwidth threshold between low and medium is 35%, while the medium to high is 70% (the point where the system enters bus saturation). The finally used workload sets are listed in Table 8.3. For the

Table 8.2: Full-system, detailed simulation parameters for the presented criticality-based priority scheme.

Core Characteristics	**Clock Frequency**	**Pipeline**	**Reorder Buffer /Scheduler**	**Branch Predictor**
	4 GHz	30 stages / 4-wide fetch / decode	128/64 Entries	Direct YAGS / indirect 256 entries
Prefetcher	H/W stride n with dynamic depth, 32 streams / core (see Section 8.3.2 for details)			
Memory Subsystem	**L1 Data & Inst. Cache**	**L2 Cache**	**Outstanding Requests**	**Memory Latency**
	64 KB, 2-way associative, 3 cycles access time, 64B blocks	16 MB, 8 ways associative, 12 cycles bank access, 64B blocks	16 Requests per Core	65ns
	Controller Organization	**DRAM**	**Controller Resources**	**Memory Bandwidth**
	2 Memory Controllers 2 Ranks per Controller 8 DRAM chips per Rank	DDR3 1333MHz 8-8-8	32 Read Queue & 32 Write Queue Entries	21.333 GB/s

evaluation, we fast-forwarded each experiment to its most representative execution phase; use the next 100M instructions to warm up the caches and memory controller structures; and then simulate the set until the slower benchmark completes 100M instructions. We only use the statistics gathered for the representative 100M instruction phase after the warming up period.

For each experiment we present as a speed-up estimation the weighted throughput:

$$\sum_{i=0}^{\#Cores} \left(\frac{IPC_i}{IPC_{i,FR-FCFS}} \right) \tag{8.1}$$

where $IPC_{i,FR-FCFS}$ is the IPC of the i-th application measured in the FR-FCFS baseline system [66] using an open-page policy memory controller. In addition, to estimate the execution fairness of every proposal, we utilize the harmonic mean of weighted speedup

Table 8.3: Randomly selected 8-core workload sets from SPEC cpu2006 for evaluation of presented scheme.

Exp. #	**Workload Sets (Core-0 → Core-7)**
1	xalancbmk, xalancbmk, omnetpp, soplex, lbm, omnetpp, wrf, zeusmp
2	bzip2, omnetpp, astar, libquantum, xalancbmk, soplex, sjeng, sjeng
3	gcc, leslie3d, zeusmp, sjeng, zeusmp, libquantum, mcf, gcc
4	gcc, namd, lbm, namd, soplex, lbm, tonto, milc
5	wrf, omnetpp, leslie3d, gamess, xalancbmk, tonto, lbm, xalancbmk
6	omnetpp, namd, GemsFDTD, leslie3d, calculix, GemsFDTD, bzip2, wrf
7	omnetpp, mcf, cactusADM, xalancbmk, mcf, omnetpp, GemsFDTD, gamess
8	gcc, omnetpp, omnetpp, xalancbmk, dealII, xalancbmk, cactusADM, libquantum
9	lbm, gamess, xalancbmk, sphinx3, mcf, soplex, omnetpp, omnetpp
10	gcc, soplex, tonto, soplex, leslie3d, libquantum, namd, astar
11	namd, xalancbmk, leslie3d, soplex, dealII, tonto, sphinx3, mcf
12	cactusADM, libquantum, libquantum, milc, gamess, mcf, omnetpp, soplex
13	omnetpp, namd, soplex, libquantum, h264ref, astar, lbm, lbm
14	soplex, xalancbmk, lbm, milc, omnetpp, perlbench, mcf, milc
15	libquantum, mcf, soplex, gromacs, omnetpp, xalancbmk, omnetpp, bwaves
16	xalancbmk, libquantum, lbm, gamess, omnetpp, mcf, xalancbmk, namd
17	hmmer, sphinx3, xalancbmk, cactusADM, libquantum, xalancbmk, zeusmp, GemsFDTD
18	GemsFDTD, wrf, gromacs, lbm, lbm, sphinx3, cactusADM, mcf
19	libquantum, astar, libquantum, sphinx3, xalancbmk, sphinx3, wrf, h264ref
20	bzip2, calculix, soplex, milc, lbm, xalancbmk, libquantum, namd
21	lbm, libquantum, lbm, mcf, cactusADM, lbm, xalancbmk, perlbench
22	leslie3d, sphinx3, xalancbmk, bzip2, h264ref, leslie3d, GemsFDTD, gobmk
23	sphinx3, sjeng, sphinx3, xalancbmk, leslie3d, mcf, soplex, xalancbmk
24	perlbench, soplex, lbm, lbm, xalancbmk, milc, libquantum, calculix
25	bwaves, leslie3d, omnetpp, xalancbmk, soplex, mcf, leslie3d, GemsFDTD
26	bwaves, libquantum, xalancbmk, namd, libquantum, libquantum, omnetpp, GemsFDTD
27	GemsFDTD, perlbench, lbm, astar, libquantum, xalancbmk, libquantum, zeusmp

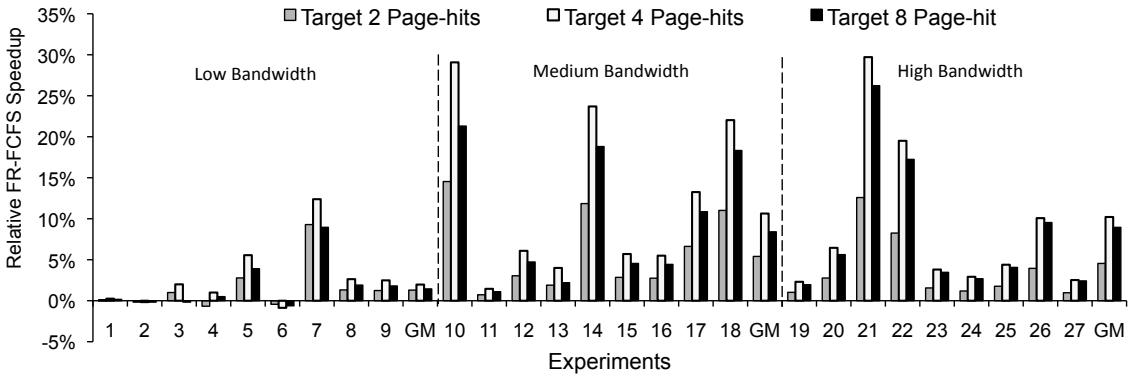


Figure 8.6: Speedup of targeting 2, 4, and 8 sequential page hits, compared to FR-FCFS.

that is:

$$Fairness = \frac{N}{\sum_{i=0}^{\#Cores} \frac{IPC_{i,alone}}{IPC_{i,shared}}} \tag{8.2}$$

where $IPC_{i,alone}$ is the IPC of the i-th application when it was executed standalone, as was previously suggested by Luo *et al.* [47].

The proposed scheme is compared against three of the most representative memory controller policies proposed in the past: a) *Parallelism-aware Batcher Scheduler* (PA-BS) [56], b) *Adaptive per-Thread Least-Attained-Service* memory scheduler (ATLAS) [39], and c) *First-Ready, First-Come-First-Served* (FR-FCFS) [66] with open-page policy. More information about the techniques can be found in Section 2.3.

8.4.1 Target Page-hit Count Sensitivity

With our system, a target page-hit count must be selected. The target page hit count indicates the maximum number of pages hits the scheme attempts. As described in Section 8.2, for 1333MHz DDR3 DRAM, the ”knee of the curve” for page-mode hits is ≈four page hits. To validate this, we evaluated targets of two, four, and eight page hits.

This is implemented as either one, two, or three column bits above the cache block in the address mapping scheme (see Figure 8.5 where a target of four is shown). The results of the simulations are shown in Figure 8.6. As expected, the target of four yields the highest system throughput. Interestingly, we found the target of two to work fairly well for low bandwidth workloads. This is somewhat expected, as low bandwidth workloads have lower bank utilization, therefore this benefit of page mode is less important. Conversely, a target of eight was most effective for high bandwidth workloads. Throughout the description and for the remaining of the evaluation we have selected to target 4 page-hits. Power consumption analysis results are included in Section 8.4.5.

8.4.2 Throughput

For the analysis we use FR-FCFS as a baseline system, and evaluated the relative improvements for PABS, ATLAS, and our proposed scheme. The results are shown in Figure 8.7. For the lowest bandwidth workloads (1 to 4) we observed no improvements over FR-FCFS. As the memory utilization increases, some reasonable gains are seen on workloads 5 and 7. For the medium bandwidth workloads, we see significant gains for our proposal over all other policies. For example, in workload set 10, we achieve more than 28% gains over all other policies. These gains are mainly due to the memory streaming workload `libquantum` as Table 8.3 shows. `Libquantum` is disruptive in systems with typical open-page mapping, where low bank level parallelism is observed. This scheme does well in this case, due to the inherently fair memory hash, where `libquantum` does not ”park” on a specific memory bank for many sequential requests. In addition, both PABS and ATLAS are effective only when the memory queues contains reasonable numbers of operations. This does not occur unless bandwidth is saturated.

Another example where the the proposed policy outperforms the other policies is Workload set 14. This workload sets contains the `mcf` benchmark that generates a

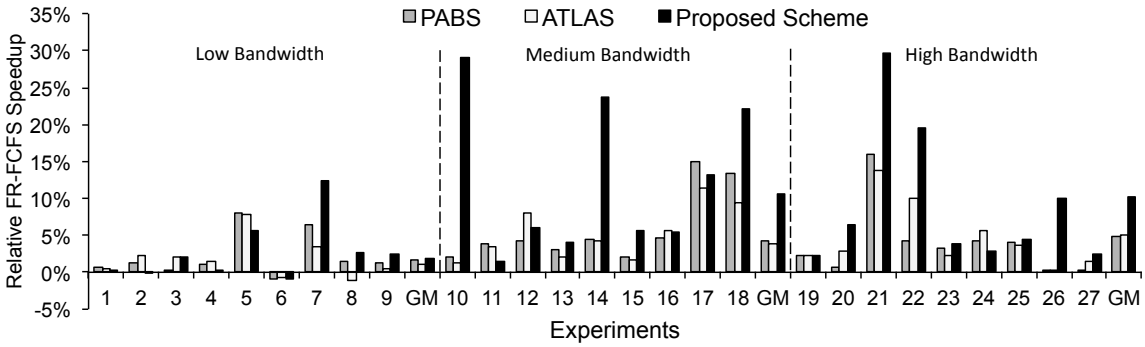


Figure 8.7: Speedup of PABS, ATLAS, and proposed scheme relative to FR-FCFS.

concurrent mix of critical demand misses and prefetch streams. The per-request priority scheme utilized in the approach enables the high priority load misses of `mcf` to be served higher priority than the less critical prefetch streams. In PABS, the scheduler is unaware of the differences in criticality within the same application grouping of all of the `mcf` requests in the same batch with equal priority. ATLAS on the other hands, treats all requests from `mcf` with a lower priority over the rest of the workloads because of it's relatively high bandwidth consumption.

For high bandwidth workloads, such as experiment sets 18, 21 and 22 from Table 8.3, more significant gains are seen for all policies. The larger amount of memory queuing enables the PABS and ATLAS policies to be effective. However, even in these cases the our policy shows higher throughput. For medium bus utilization we achieved a 11.1%, 7.3% and 6.4% speedup improvement than FCFR, PABS and ATLAS, respectively. Finally, for high bus utilization the speedup improvements was 10.2%, 5.4%, 5.7% over FCFR, PABS and ATLAS, respectively.

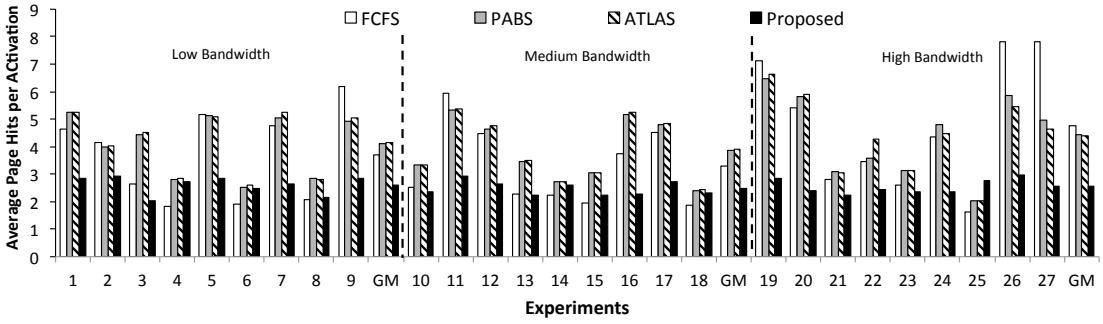


Figure 8.8: Average number of page-hits per page activation for all schemes

8.4.3 Page-hits per Page activation

Figure 8.8 includes the average number of page-hits per page activation that each scheme achieved in our system. As the figure shows, although the proposed scheme is restricted to a maximum of 4 page hits per activation, it managed to achieve an average number very close to 3 page-hits per activation. As this average number includes all the possible page-hits (demand reads + prefetches) it is evident that the prefetch guided page-mode policy was really effective in finding and extracting these useful page-hits. Its efficiency is evident by combining the average page-hits numbers with the throughput improvement for the proposed scheme of Figure 8.7. The proposed scheme provided the best throughput improvements in most of the cases while maintaining this small number of targeted page accesses.

From Figure 8.8, we can see that FR-FCFS achieves the highest number of page-hits due to its unfair policy to give higher priority to accesses to already open pages. Such unfairness is the reason that even though FR-FCFS had the highest number of page-hits, it also had the worst performance over all the other evaluated schemes. ATLAS and PABS feature a slightly smaller average number of page-hits over the FR-FCFS case. Effectively, ATLAS and PABS, due to their priority scheme of reordering memory

request in the memory queue, interchange number of page-hits for fairness and throughput improvements. The results show that just targeting row-buffer locality is not enough to improve multicore runs efficiency and fairness. Moreover, the proposed address mapping scheme can effectively provide throughput improvements while maintaining a small number of page-hits per activation close to 3. This is a proof that fixing the fairness directly on the DRAM address mapping scheme rather than in the memory queue priority scheme can achieve better performance while extracting most of the benefits of page-mode, as the motivation section described.

There are a number of experiments in Figure 8.8 that FR-FCFS, ATLAS and PABS achieved an average number of page-hits lower than the proposed scheme, especially for the cases of "Medium bandwidth" use. In these cases, the proposed prefetch directed page-mode policy and the address mapping scheme were able to extract better spatial locality from the DRAM row-buffer than the other schemes. ATLAS and PABS schemes significantly hurt the applications requests' spatial locality because they also have to fix the execution fairness by reordering memory requests in the memory controller. Finally, FR-FCFS as with the case of the other schemes, cannot extract many page hits because in the case of "Medium Bandwidth" there are not many requests queued up in the memory controller from the same application to find many page-hits to an already open-page. As the bandwidth use saturates and requests remain for more time in the memory controller queue, these schemes are able to find more page-hits opportunities ("High Bandwidth" cases from Figure 8.8)

8.4.4 Fairness

Figure 8.9 shows the fairness improvement of all schemes relative to FR-FCFS baseline system using the harmonic mean of weighted speedup. It is important to note that the throughput gains we achieve are accompanied with improvements in the fairness.

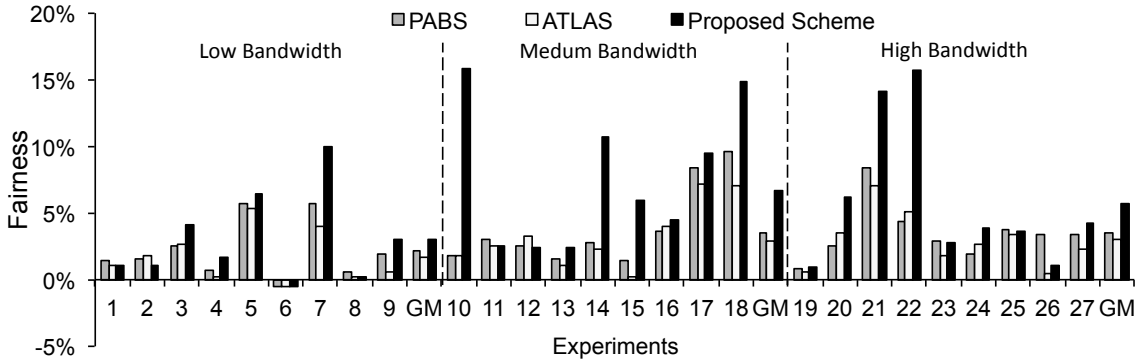


Figure 8.9: Execution fairness improvements compared to FR-FCFS scheme.

This is expected as the throughput gains are a result of alleviating unresolveable conflict cases associated with row buffer starvation. Essentially, the proposed scheme matches the throughput gains in cases without row buffer conflicts while significantly improving cases where row buffer conflicts exist.

As explained by Kim [39], ATLAS is less fair than PABS, since ATLAS targets throughput over fairness (interestingly we saw similar throughput for both algorithms in the experiments). Our scheme improves fairness up to 15% with an overall improvement of 6.4%, 2.8% and 2.1% over medium bus utilization, and 5.74%, 3.48%, 2.87% over high utilization for FCFR, PABS and ATLAS, respectively.

8.4.5 DRAM Energy Consumption

To evaluate if the throughput and fairness gains do not adversely effect the system energy consumption, we show in Figure 8.10 the DRAM energy of the PABS, ATLAS, and the our proposal policies relative to the FR-FCFS policy. To estimate the power consumption we used the Micron power calculator [51]. The geometric mean of the relative energy across all experiments of the policies is approximately the same as FR-

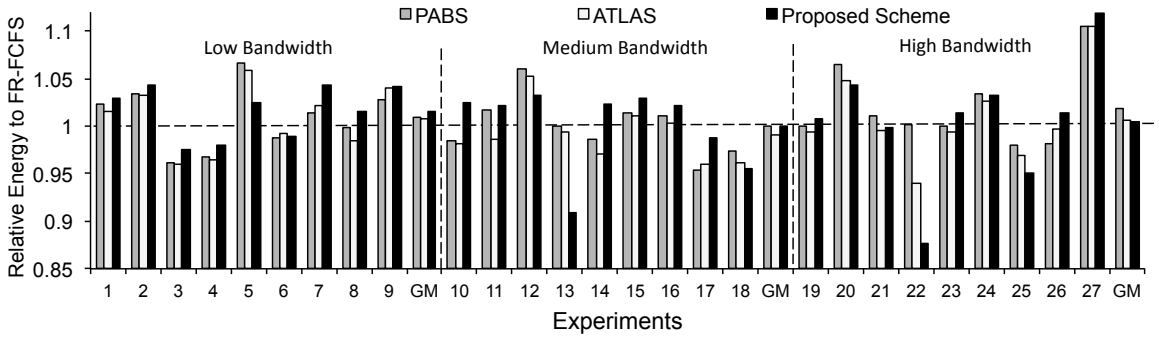


Figure 8.10: DRAM energy improvements relative to FR-FCFS scheme.

FCFS. Both PABS and Minimal increase energy by a small fraction of 0.8% and 0.6%, respectively. The ATLAS approach resulted in essentially no energy change from FR-FCFS (increase of 0.08%). All of the results are essentially a balance of the decrease in page mode hits (resulting in high DRAM activation power) and the increase in system performance (decreasing runtime). Note, decreasing runtime has the effect of decreasing background power effect on energy. In addition, these results are based on the DDR3 row-buffer size. With the proposed policy, a DRAM device could be designed with much smaller row-buffers, which would result in an energy reduction for the proposed olicy as compared to current practices.

8.5 Summary

This chapter presents an efficient and fair memory scheduling policy. As page mode gains can be realized with a relatively small number of page accesses for each page activation, a policy with "just enough" page-mode accesses per bank can effectively eliminates the negative aspects of page mode, such as starvation and row buffer conflicts, while maintaing most of its gains. Using this fair policy, the presented scheme is able to build intuitive memory scheduler priority policies based strictly on age and request

criticality. These request attributes are derived through monitoring program Memory-level Parallelism (MLP) and request stream information within data prefetch engine.

Overall the scheme is effective in concurrently improving throughput and fairness across a wide range of memory utilization levels. It is particularly effective, compared to prior work, in improving workload combinations that contain streaming memory references (up to 30% improvements in throughput and 15% in fairness). Compared to prior work, thread based priority information is not needed (or helpful), which enables workloads with requests of multiple priorities to be efficiently scheduled. In addition, no coordination between multiple memory controllers or operating system interaction is required. This alleviate overall system complexity, enabling other components of the system to be optimized.

Chapter 9

Summary and Future Work

9.1 Summary

This dissertation focuses on the importance of memory-subsystem resource management in current and future many-core designs. Such an efficient management of memory has become more critical than ever before because of the increasing memory latency, increasing off-chip aggregated memory bandwidth demands and the decreasing on-chip cache capacity size available per core/thread. This dissertation focuses on analyzing the challenges of managing the memory subsystem in many-core designs, and presents cost-effective solutions to improve many-core systems performance and execution fairness.

The dissertation presents efficient solutions to: a) provide mechanisms that can capture the resource requirements for each application for every different shared resource, b) describe microarchitecture-level mechanisms that can enforce specific shared resource allocations, and finally, c) propose resource allocation algorithms and/or policies that can manage the resource allocation mechanisms based on the applications requirements.

Chapter 4 presents low overhead, non-invasive, hardware profiler mechanisms based on Mattson's stack distance algorithm that can effectively project applications' memory resource requirements and memory sharing behavior. The proposed mechanisms are used throughout the dissertation as efficient means of profiling applications requirements and can effectively guide the proposed dynamic resource managing schemes of Chapters 5-8.

Chapter 5 describes a dynamic last-level cache partitioning scheme for the CMP-DNUCA architecture, the *Bank-aware Cache Partitioning*. The proposed scheme is consistent with the current industry trends, that is aware of the banking structure of the L2 cache. The provided solution is able to provide significant reductions in misses over static partitioning schemes; achieving performance improvements close to the reported one from previous more theoretical implementations that assume simplified, single-bank, shared last-level caches.

Chapter 6 presents a *Bandwidth-aware* resource managing scheme for large CMP systems. The technique proposes a system-wide optimization of memory-subsystem resource allocation and job scheduling that aims to achieve overall system throughput optimization in large, multi-level CMP systems. It first solves the local inter-chip contention on each individual chip and following that, seeks for additional optimizations, in terms of memory bandwidth use and/or cache capacity requirements, both inside and outside a single chip in the system. Such bandwidth-aware scheme is able to achieve a reduction of 18% in memory bandwidth along with an average 8.5% increase in IPC with marginal additional hardware overhead over single-chip optimization policies.

Chapter 7 describes a *Quasi-partitioning* scheme for last-level caches. The scheme combines the memory-level parallelism (MLP), cache friendliness and cache interference sensitivity of competing applications, to efficiently manage the shared cache capacity. The addition of MLP information demonstrated that only focusing on the reduction of the overall number of misses does not always lead to higher performance, as MLP can hide the latency penalty of a significant number of misses in out-of-order execution. The proposed scheme outperforms previous schemes my taking MLP and application's memory sharing characteristics into account.

Finally, Chapter 8 describes a read latency criticality-based, memory controller requests priority scheme for many-core designs. The scheme is based on the design

philosophy that in order to improve overall system performance and fairness, all the components of memory hierarchy have to be coordinated. The key observations is that DRAM page-mode access gains are realized with only a small number accesses per page activation and this page-mode opportunities are usually the results of memory accesses generated through prefetch operations. Based on this insight, the scheme directs the memory controller priority scheme along with DRAM page mode policy using prefetcher's unit meta-data to optimize memory resource use. Such approach demonstrated significant gains in throughput and fairness over the best prior proposals for medium and high memory utilization levels.

9.2 Future Work

9.2.1 Multi-chip / multi-socket QoS scheme with capacity planning

Large computational servers and data-centers with multiple, hierarchically interconnected (either tightly through memory or loosely through network) chips/sockets typically need to be able to share their resources in a fair way or according to a specific service level agreement (SLA). For those systems, the ability to scale any QoS solution to a large number of chips/boards/servers is crucial for the scheme's effectiveness and applicability. An extension of the *Bandwidth-aware Memory-subsystem Resource Management* scheme could be the application of a QoS scheme over a multi-chip / multi-socket environment. The existing so far QoS proposals, restrict their scope on a single multicore system and therefore lose the opportunity of investigating solutions and optimizations beyond a single chip in multichip/multisocket systems. The scheme could target the use of the minimum number of CMP cores across the whole system in order to execute all applications with specific QoS service levels. Various job admission control schemes can also be investigated in such environment that could decide, if and where exactly, the system can handle the additional load without affecting other applications'

QoS targets.

9.2.2 Extension of Resource Managing schemes for fairness/QoS

The memory-resource partitioning schemes presented in this dissertation focused on improving performance and fairness among the demand requests of the applications to the memory. Such approach focuses on the demand read latency improvement. The presented schemes can be extended to include prefetch data along with the demand reads, allowing the scheme to control both directions of memory traffic on the shared last-level cache. In addition, the Quasi-partitioning scheme and be easily extended to target system fairness directly and/or a QoS scheme. The scheme provides a direct way to estimate the impact of changes in resource allocations to final, measurable performance. Using this information the resource allocation algorithm can be tuned to target either a fair or a specific percentage of IPC improvement per application to respect a specific QoS service level.

9.2.3 Bandwidth-aware page allocation memory controller policy

DRAM devices use the row-buffer as a mean to improve demand reads latency and DRAM power consumption of successive accesses to the same page. As CMP designs move to higher number of cores using almost the same number of physical chip pins per generation, the pressure on the IO bandwidth requirements is drastically increasing. Maintaining a high level of memory bandwidth (throughput) to such designs becomes one of the most challenging tasks. The memory controller priority scheme described in this dissertation can be extended to target higher bandwidth with better use of the DRAM memory characteristics and restrictions. For example, a different, dynamic memory address mapping scheme could target to maintain the highest number of open useful pages by remapping and distributing “hot” pages to specific DRAM ranks/banks while collapsing

the “cold” ones on a different set of ranks/banks. In addition, applications with high bandwidth demands could be assigned a number of dedicated ranks that can service them in parallel, while the rest, lower bandwidth applications can share some ranks/banks. Of course such approaches require to modify the way the operating system assigns physical addresses to memory pages and can potentially need to break “hot” pages in multiple, smaller pages that are stored in different ranks/banks in the DRAM.

Bibliography

[1] Paul Barford and Mark Crovella. Generating representative web workloads for network and server performance evaluation. In *Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, SIGMETRICS '98/PERFORMANCE '98, pages 151–160, New York, NY, USA, 1998. ACM.

[2] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37:164–177, October 2003.

[3] Bradford M. Beckmann and David A. Wood. Managing wire delay in large chip-multiprocessor caches. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 319–330, Washington, DC, USA, 2004. IEEE Computer Society.

[4] Ramazan Bitirgen, Engin Ipek, and Jose F. Martinez. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 318–329, Washington, DC, USA, 2008. IEEE Computer Society.

[5] Doug Burger, James R. Goodman, and Alain Kägi. Memory bandwidth limitations of future microprocessors. In *Proceedings of the 23rd annual international symposium on Computer architecture*, ISCA '96, pages 78–89, New York, NY, USA, 1996. ACM.

[6] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. Predicting Inter-Thread cache contention on a chip multi-processor architecture. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 340–351, Washington, DC, USA, 2005. IEEE Computer Society.

[7] Jichuan Chang and Gurindar S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *Proceedings of the 21st annual international conference on Supercomputing*, ICS '07, pages 242–252, New York, NY, USA, 2007. ACM.

[8] Yuan Chou, Brian Fahs, and Santosh Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. In *Proceedings of the 31st annual international symposium on Computer architecture*, ISCA '04, pages 76–, Washington, DC, USA, 2004. IEEE Computer Society.

[9] Intel Corporation. First the tick, now the tock: Next generation intel microarchitecture (Nehalem), white paper, http://www.intel.com/technology/architecture-silicon/next-gen/whitepaper.pdf.

[10] Jack Doweck. Inside intel core microarchitecture and smart memory access, 2006. http://www.influentmotion.com/Server White Paper.pdf.

[11] The Apache Software Foundation. Apache HTTP server version 2.0 documentation, 2011, http://httpd.apache.org/docs/2.0/.

[12] Q. S. Gao. The chinese remainder theorem and the prime memory system. In *Proceedings of the 20th annual international symposium on Computer architecture*, ISCA '93, pages 337–340, New York, NY, USA, 1993. ACM.

[13] P. Glaskowsky. High-end server chips breaking records, "http://news.cnet.com/8301-13512_3-10321740-23.html", Aug. 2009.

[14] Fei Guo, Hari Kannan, Li Zhao, Ramesh Illikkal, Ravi Iyer, Don Newell, Yan Solihin, and Christos Kozyrakis. From chaos to QoS: case studies in CMP resource management. *SIGARCH Comput. Archit. News*, 35:21–30, March 2007.

[15] Wim Heirman. SPLASH-2 for solaris on sparc/simics, http://trappist.elis.ugent.be/ wheirman/simics/splash2/.

[16] John L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[17] Enric Herrero, José González, and Ramon Canal. Elastic cooperative caching: an autonomous dynamically adaptive memory hierarchy for chip multiprocessors. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 419–428, New York, NY, USA, 2010. ACM.

[18] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Trans. Comput.*, 38:1612–1630, December 1989.

[19] R. Ho, K.W. Mai, and M.A. Horowitz. The future of wires. *Proceedings of the IEEE*, 89(4):490 –504, April 2001.

[20] Lisa R. Hsu, Steven K. Reinhardt, Ravishankar Iyer, and Srihari Makineni. Communist, utilitarian, and capitalist cache policies on cmps: caches as a shared resource. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, PACT '06, pages 13–22, New York, NY, USA, 2006. ACM.

[21] Jaehyuk Huh, Doug Burger, and Stephen W. Keckler. Exploring the design space of future cmps. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, PACT '01, pages 199–210, Washington, DC, USA, 2001. IEEE Computer Society.

[22] I. Hur and C. Lin. Feedback mechanisms for improving probabilistic memory prefetching. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pages 443 –454, feb. 2009.

[23] Ravi Iyer. CQoS: a framework for enabling QoS in shared caches of CMP platforms. In *Proceedings of the 18th annual international conference on Supercomputing*, ICS '04, pages 257–266, New York, NY, USA, 2004. ACM.

[24] Bruce Jacob, Spencer Ng, and David Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[25] J. Jaehyuk Huh, C. Changkyu Kim, H. Shafi, L. Lixin Zhang, D. Burger, and S.W. Keckler. A NUCA substrate for flexible CMP cache sharing. *Parallel and Distributed Systems, IEEE Transactions on*, 18(8):1028 –1040, 2007.

[26] Aamer Jaleel, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Simon Steely, Jr., and Joel Emer. Adaptive insertion policies for managing shared caches. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 208–219, New York, NY, USA, 2008. ACM.

[27] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. High performance cache replacement using re-reference interval prediction (RRIP). In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 60–71, New York, NY, USA, 2010. ACM.

[28] JEDEC Committee JC-42.3. JESD79-3D, September 2009.

[29] Yunlian Jiang, Xipeng Shen, Jie Chen, and Rahul Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *Proceedings*

of the 17th international conference on Parallel architectures and compilation techniques, PACT '08, pages 220–229, New York, NY, USA, 2008. ACM.

[30] Ron Kalla, Balaram Sinharoy, William J. Starke, and Michael Floyd. POWER 7: IBM's next-generation server processor. *IEEE Micro*, 30:7–15, March 2010.

[31] Tejas Karkhanis and J. E. Smith. A day in the life of a data cache miss. In *In Workshop on Memory Performance Issues*, 2002.

[32] Dimitris Kaseridis, Jeffrey Stuecheli, Jian Chen, and Lizy. K. John. A bandwidth-aware memory-subsystem resource management using non-invasive resource profilers for large CMP systems. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1 –11, 2010.

[33] Dimitris Kaseridis, Jeffrey Stuecheli, and Lizy K. John. Bank-aware dynamic cache partitioning for multicore architectures. In *Proceedings of the 2009 International Conference on Parallel Processing*, ICPP '09, pages 18–25, Washington, DC, USA, 2009. IEEE Computer Society.

[34] R. E. Kessler, M. D. Hill, and D. A. Wood. A comparison of trace-sampling techniques for multi-megabyte caches. *IEEE Trans. Comput.*, 43:664–675, June 1994.

[35] R. E. Kessler, R. Jooss, A. Lebeck, and M. D. Hill. Inexpensive implementations of set-associativity. In *Proceedings of the 16th annual international symposium on Computer architecture*, ISCA '89, pages 131–139, New York, NY, USA, 1989. ACM.

[36] Kevin Kilbuck. Main memory technology direction. In *Microsoft WinHEC*, 2007.

[37] Changkyu Kim, D. Burger, and S.W. Keckler. Nonuniform cache architectures for wire-delay dominated on-chip caches. *Micro, IEEE*, 23(6):99 – 107, nov.-dec. 2003.

[38] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 111–122, Washington, DC, USA, 2004. IEEE Computer Society.

[39] Yoongu Kim, Dongsu Han, O. Mutlu, and M. Harchol-Balter. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1 –12, jan. 2010.

[40] Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *the future proceedings of the 43rd annual IEEE/ACM International Symposium on Microarchitecture*, 2010.

[41] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th International Symposium on Computer Architecture*, pages 81–87, 1981.

[42] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden. IBM POWER6 microarchitecture. *IBM Journal of Research and Development*, 51(6):639 –662, 2007.

[43] Chang Joo Lee, Onur Mutlu, Veynu Narasiman, and Yale N. Patt. Prefetch-aware DRAM controllers. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, pages 200–209, 2008.

[44] Yingmin Li, B. Lee, D. Brooks, Zhigang Hu, and K. Skadron. CMP design space exploration subject to physical constraints. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pages 17 – 28, 2006.

[45] Wei-Fen Lin, S.K. Reinhardt, and D. Burger. Designing a modern memory hierarchy with hardware prefetching. *IEEE Transactions on Computers*, 50(11):1202 –1218, November 2001.

[46] Chun Liu, Anand Sivasubramaniam, and Mahmut Kandemir. Organizing the last line of defense before hitting the memory wall for CMPs. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, HPCA '04, pages 176–, Washington, DC, USA, 2004. IEEE Computer Society.

[47] Kun Luo, Jayanth Gummaraju, and Manoj Franklin. Balancing throughput and fairness in SMT processors. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2001.

[48] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

[49] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet: A general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comput. Archit. News*, 33, 2005.

[50] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Syst. J.*, 9:78–117, June 1970.

[51] Micron Technologies, Inc. DDR3 SDRAM system-power calculator, revision 0.1, 2007.

[52] Miquel Moreto, Francisco J. Cazorla, Alex Ramirez, and Mateo Valero. MLP-aware dynamic cache partitioning. In *Proceedings of the 3rd international conference on High performance embedded architectures and compilers*, HiPEAC'08, pages 337–352, Berlin, Heidelberg, 2008. Springer-Verlag.

[53] Thomas Moscibroda and Onur Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, pages 18:1–18:18, 2007.

[54] Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. Optimizing nuca organizations and wiring alternatives for large caches with CACTI 6.0. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 3–14, Washington, DC, USA, 2007. IEEE Computer Society.

[55] Onur Mutlu and Thomas Moscibroda. Stall-Time fair memory access scheduling for chip multiprocessors. In *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, pages 146 –160, 2007.

[56] Onur Mutlu and Thomas Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *ISCA '08: Proceedings of the 35th Annual International Symposium on Computer Architecture*, pages 63–74, 2008.

[57] Kyle J. Nesbit, Nidhi Aggarwal, James Laudon, and James E. Smith. Fair queuing memory systems. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 208–222, 2006.

[58] Kyle J. Nesbit, Miquel Moreto, Francisco J. Cazorla, Alex Ramirez, Mateo Valero, and James E. Smith. Multicore resource management. *IEEE Micro*, 28:6–16, May

2008.

[59] SPEC Organization. SPEC CPU2000 benchmark suit, http://ftp.spec.org/cpu2000/.

[60] SPEC Organization. SPEC JBB2005 benchmark suit, http://www.spec.org/jbb2005/.

[61] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. Adaptive insertion policies for high performance caching. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 381–391, New York, NY, USA, 2007. ACM.

[62] Moinuddin K. Qureshi, Daniel N. Lynch, Onur Mutlu, and Yale N. Patt. A case for MLP-aware cache replacement. In *Proceedings of the 33rd annual international symposium on Computer Architecture*, ISCA '06, pages 167–178, Washington, DC, USA, 2006. IEEE Computer Society.

[63] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 423–432, Washington, DC, USA, 2006. IEEE Computer Society.

[64] Nauman Rafique, Won-Taek Lim, and Mithuna Thottethodi. Architectural support for operating system-driven CMP cache management. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, PACT '06, pages 2–12, New York, NY, USA, 2006. ACM.

[65] Parthasarathy Ranganathan and Norman Jouppi. Enterprise IT trends and implications for architecture research. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 253–256, Washington, DC, USA, 2005. IEEE Computer Society.

[66] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. Memory access scheduling. In *ISCA '00: Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 128–138, 2000.

[67] Brian M. Rogers, Anil Krishna, Gordon B. Bell, Ken Vu, Xiaowei Jiang, and Yan Solihin. Scaling the bandwidth wall: challenges in and avenues for CMP scaling. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 371–382, 2009.

[68] André Seznec and Jacques Lenfant. Odd memory systems may be quite interesting. In *Proceedings of the 20th annual international symposium on Computer architecture*, ISCA '93, pages 341–350, New York, NY, USA, 1993. ACM.

[69] John Shen and Mikko Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*. McGraw-Hill Education.

[70] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. POWER5 system microarchitecture. *IBM J. Res. Dev.*, 49:505–521, July 2005.

[71] Alan Jay Smith. Cache memories. *ACM Computing Surveys (CSUR)*, 14:473–530, September 1982.

[72] Shekhar Srikantaiah, Mahmut Kandemir, and Mary Jane Irwin. Adaptive set pinning: managing shared caches in chip multiprocessors. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, pages 135–144, New York, NY, USA, 2008. ACM.

[73] Harold S. Stone, John Turek, and Joel L. Wolf. Optimal partitioning of cache memory. *IEEE Trans. Comput.*, 41:1054–1068, September 1992.

[74] Jeffrey Stuecheli, Dimitris Kaseridis, David Daly, Hillery C. Hunter, and Lizy K. John. The Virtual Write Queue: coordinating DRAM and last-level cache policies. In *ISCA '10: Proceedings of the 37th annual international symposium on Computer architecture*, pages 72–82, 2010.

[75] Kshitij Sudan, Niladrish Chatterjee, David Nellans, Manu Awasthi, Rajeev Balasubramonian, and Al Davis. Micro-pages: increasing DRAM efficiency with locality-aware data placement. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 219–230, 2010.

[76] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 28:7–26, April 2004.

[77] G. Edward Suh, Srinivas Devadas, and Larry Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, HPCA '02, Washington, DC, USA, 2002. IEEE Computer Society.

[78] Dennis Sylvester and Kurt Keutzer. Getting to the bottom of deep submicron II: a global wiring paradigm. In *Proceedings of the 1999 international symposium on Physical design*, ISPD '99, pages 193–200, New York, NY, USA, 1999. ACM.

[79] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM J. Res. Dev.*, 46:5–25, January 2002.

[80] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kagi, Felix H. Leung, and Larry Smith. Intel virtualization technology. *Computer*, 38:48–56, May 2005.

[81] Friedrich von Wieser. *Der naturliche Werth [Natural Value]*. Book I. 1889.

[82] M. V. Wilkes. Readings in computer architecture, Chapter: Slave memories and dynamic storage allocation. pages 371–372. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[83] Maurice V. Wilkes. The memory gap and the future of high performance memories. *SIGARCH Computer Architecture News*, 29:2–7, March 2001.

[84] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd annual international symposium on Computer architecture*, ISCA '95, pages 24–36, New York, NY, USA, 1995. ACM.

[85] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Computer Architecture News*, 23:20–24, March 1995.

[86] Yuejian Xie and Gabriel H. Loh. PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 174–183, New York, NY, USA, 2009. ACM.

[87] Zhao Zhang, Zhichun Zhu, and Xiaodong Zhang. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, MICRO 33, pages 32–41, 2000.

[88] Li Zhao, Ravi Iyer, Ramesh Illikkal, Jaideep Moses, Srihari Makineni, and Don Newell. Cachescouts: Fine-grain monitoring of shared caches in CMP platforms. In *Proceedings of the 16th International Conference on Parallel Architecture and*

Compilation Techniques, PACT '07, pages 339–352, Washington, DC, USA, 2007. IEEE Computer Society.

[89] Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar. Dynamic tracking of page miss ratio curve for memory management. In *Proceedings of the 11th international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-XI, pages 177–188, New York, NY, USA, 2004. ACM.

[90] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the 15th international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '10, pages 129–142, New York, NY, USA, 2010. ACM.